université de BORDEAUX

Thèse présentée pour obtenir le grade de

Docteur de l'Université de Bordeaux

École doctorale de mathématiques et d'informatique
Spécialité informatique

Par Paul-Antoine Arras

Ordonnancement d'applications dynamiques à flux de données pour les MPSoC embarqués hybrides comprenant des unités de calcul programmables et des accélérateurs matériels

Sous la direction de : Emmanuel Jeannot
(co-directeur : Samuel Thibault)

Soutenue le mardi 3 février 2015

Membres du jury :

M. Pierre Boulet	Professeur, Université Lille 1	Rapporteur
M. Daniel Étiemble	Professeur, Université Paris Sud	Rapporteur
M. Didier Fuin	STMicroelectronics Grenoble	Encadrant
M. Alain Girault	Directeur de recherche, Inria	Examinateur
M. Pascal Guitton	Professeur, Université de Bordeaux	Président
M. Emmanuel Jeannot	Directeur de recherche, Inria	Directeur de thèse
M. Samuel Thibault	Maître de conférences, Université de Bordeaux	Co-directeur de thèse

Titre : Ordonnancement d'applications dynamiques à flux de données pour les MPSoC embarqués hybrides comprenant des unités de calcul programmables et des accélérateurs matériels.

Résumé : Bien que de nombreux appareils numériques soient aujourd'hui capables de lire des contenus vidéo en temps réel et d'offrir une restitution de grande qualité, le décodage vidéo dans les systèmes embarqués n'en est pas pour autant devenu une opération anodine. En effet, les codecs récents tels que H.264 et HEVC sont d'une complexité telle que le recours à des architectures mixtes logiciel/matériel est presque incontournable. Or les plateformes de ce type sont notoirement difficiles à programmer efficacement.

Cette thèse relève le défi du développement d'applications à flux de données pour les cibles embarquées hybrides et de leur exécution efficace, et propose plusieurs contributions. La première est une extension des heuristiques d'ordonnancement de liste pour tenir compte des contraintes mémorielles. La seconde est un modèle d'exécution à flot de données compatible avec la plupart des modèles existants et avec une large classe de plateformes matérielles, ainsi qu'un ordonnanceur dynamique. Enfin, de nombreux développements ont été menés sur une architecture réelle de STMicroelectronics pour démontrer la faisabilité de l'approche.

Mots-clés : ordonnancement ; flot de données ; flux de données ; modèle d'exécution ; architectures hétérogènes ; architectures hybrides ; systèmes embarqués.

Title : Scheduling of dynamic streaming applications on hybrid embedded MPSoCs comprising programmable computing units and hardware accelerators.

Abstract : Although numerous electronic devices are nowadays able to play video contents in real time and offer high-quality reproduction, video decoding in embedded systems has not become a trivial process yet. As a matter of fact, recent codecs such as H.264 and HEVC exhibit such a complexity that resorting to mixed software-hardware architecture is almost unavoidable. However, programming efficiently this kind of platforms is well-known to be tricky.

This thesis addresses the issue of developing streaming applications for hybrid embedded targets and executing them efficiently, and proposes several contributions. The first one is an extension of the classical list-scheduling heuristics to take memory constraints into account. The second one is a dataflow execution model compatible with most existing models and with a large set of hardware platforms, as well as a dynamic scheduler. Lastly, numerous developments have been carried out on a real-world architecture from STMicroelectronics so as to demonstrate the feasibility of the approach.

Keywords : scheduling ; dataflow ; streaming ; execution model ; heterogeneous architectures ; hybrid architectures ; embedded systems.

Unité de recherche : Équipe-projet Inria Runtime, Centre de recherche Inria Bordeaux Sud-Ouest, 200 avenue de la Vieille Tour, 33 405 Talence Cedex.

« Chaque chose devrait être
aussi simple que possible,
mais pas davantage. »

Albert Einstein,
cité par Shapiro [79].

Remerciements

Mes remerciements vont en premier lieu à mes directeurs de thèse, Emmanuel Jeannot et Samuel Thibault côté Inria, et à mes encadrants, Didier Fuin et Arthur Stoutchinin côté STMicroelectronics, d'abord pour avoir accepté de me prendre en thèse, puis pour m'avoir toujours soutenu et écouté pendant toute la durée de celle-ci, surtout dans les moments difficiles – et il y en a eu ! Je leur dois également des remerciements pour m'avoir laissé une grande liberté dans mon travail de recherche, tout en ayant su me guider chaque fois que le besoin s'en faisait sentir – équilibre difficile à trouver, s'il en est ! Je remercie ensuite les membres de mon jury : Pierre Boulet et Daniel Étiemble, qui ont en outre accepté de rapporter cette thèse ; Alain Girault qui, bien que n'étant pas rapporteur, a fourni de nombreux commentaires fort constructifs ; et enfin Pascal Guitton qui a bien voulu en assumer la présidence.

J'aimerais ensuite remercier toutes les personnes qui m'ont permis d'effectuer mon travail dans de bonnes conditions – entendre par là, non pas qui m'ont permis d'être plus productif, mais qui m'ont permis de me sentir mieux dans ce que je faisais. Je commencerai par les plus proches physiquement, c'est-à-dire la fine équipe de l'*open-space* Runtime, et en particulier du carré VIP. Parmi eux, les anciens : Sylvain, sa vision fonctionnelle et son militantisme contagieux ; Bertrand, ses blagues corses ou basques (selon les circonstances) et l'approche pyramidale du sport ; Cyril B., fervent défenseur de Xchat et LibreOffice ; Cyril R. pour les discussions interminables sur le travail libre. Les occupants du carré VIP au moment de mon départ : François, notre libriste national, avec qui j'ai partagé ce parcours du combattant qu'est la thèse – et bien d'autres choses… ; Marc, qui a toujours su se montrer à la hauteur de mes attentes – voire au-delà ; Chris, et ses calembours qui cartonnent ; Nicolas et sa vision pessimiste mais néanmoins fort constructive ; Théo, avec son arme de punition massive contre les blagues indigentes. De l'autre côté de l'*open-space*, j'aimerais remercier : Terry, qui a ouvert mon oreille à de nouveaux horizons musicaux ; Samuel, fin connaisseur de La Grange et jamais le dernier pour les soirées arrosées ; Thomas, pour ses conseils avisés en vue de la soutenance ; James et Adèle, moins exubérants que le reste de la clique, mais tout aussi jovials.

Toujours par ordre de proximité, viennent ensuite les autres membres de Runtime. J'aimerais d'abord remercier : Emmanuel, Denis et Guillaume pour les soirées jeux mémorables ; Brice qui, bien que n'étant pas, la plupart du temps, physiquement dans l'*open-space*, en porte l'esprit trolleur jusque dans son bureau. Je remercie aussi Andra, Nathalie, Pierre-André, Marie-Christine, Alexandre, Olivier et, bien sûr, Raymond, le charismatique chef d'équipe, pour m'avoir accueilli chaleureusement au labo.

Avant de quitter le cadre professionnel, je voudrais remercier l'AGOS pour avoir fourni et entretenu avec le plus grand soin notre principal outil de travail : le baby-foot. À cet égard, je me dois de

remercier l'ensemble de l'équipe Hiepacs pour leur contribution soutenue, et plus particulièrement : Emmanuel, pour les commentaires taquins – vexants, diront les mauvaises langues – pendant les matchs ; Julien, pour les gamelles imaginaires ; Damien et Aurel, pour les scènes de ménage derrière les cannes ; François, pour les coups de hachoir ; Elmer, pour sa mauvaise foi et son arrogance merveilleuses. Je remercie du fond du cœur mes infortunés partenaires, qui ont accepté de jouer avec moi malgré mon incompétence flagrante. Pour ce qui est d'Hiepacs, je remercie également Abdou et Mathieu pour les discussions informelles autour d'un verre ou d'un bon repas. À ce propos, je remercie également toute l'équipe du RU du Haut-Carré pour leur bonne humeur quotidienne et pour le solide soutien apporté à nos estomacs insatiables le midi ; de même pour la cafétéria Inria et les petits-déjeuners.

Hors Inria, je souhaite remercier les nombreux colocataires avec qui j'ai eu la chance de partager des moments de vie, par ordre chronologique : Nicolas, qui m'a apporté un soutien fort appréciable à mon arrivée à Grenoble, ce pour quoi je lui suis profondément reconnaissant ; Christine ; Fred ; JB, à qui, il me semble, je dois le qualificatif de « nazi de la télé » ; Gaylord, qui m'a vendu du rêve avec ses potes vendéens ; Juliette ; Pierre ; Plou ; PM ; Céline ; Yuran et son adorable boule de poils ; Sophie et son abominable rongeur aux oreilles tombantes ; Théo, la personne la plus joviale que j'ai rencontrée à Bordeaux, avec qui j'ai pu avoir des échanges très positifs sur les difficultés de mener une thèse en entreprise ; Coline, toujours très sport ; Aurélie, gentiment délurée, qui a rythmé nos vendredis soirs à la bière et favorisé les échanges inter-colocs ; Benoît ; Lucie ; et nos voisins de maisonnée : Benoît, Pierre, Clémence, Vincent, Maëva, Alexy et Marie. En outre, je remercie tous ceux avec qui j'ai passé de bons moments à Grenoble : Jun-Young, pour les nombreuses sorties ski démentielles ; Betül ; Yeter, l'organisatrice en chef de soirées ; Lionel ; Jérôme et ses lettres d'excuse « canoniques » ; Maud ; Jander ; Ronan, à qui je dois mon unique expérience de ski nocturne ; Victor ; Ozan ; Molka ; ainsi que tous ceux, trop nombreux pour être cités, que j'ai eu le plaisir de côtoyer lors de soirées ou sorties.

Pour conclure, je remercie toute ma famille pour leur soutien permanent pendant ces trois ans, et en particulier mes parents et ma sœur qui sont venus de loin pour ma soutenance.

Avant-propos

L'électronique embarquée imprègne nos vies. C'est un fait. Il n'y a qu'à observer l'essor des téléphones portables, aussi bien d'un point de vue commercial qu'au regard de l'usage qui en est fait, pour s'en convaincre. Et pourtant, la plupart des gens soit n'ont aucune idée de ce que le terme « électronique embarquée » recouvre, soit en ont une réductrice. Quand on entend « électronique embarquée », on pense bien souvent robotique, aérospatiale, aviation, automobile,… mais certainement pas téléphonie mobile, et encore moins décodage vidéo. « On » c'était moi jusqu'à il y a un peu moins de cinq ans, en 2010. À l'époque, j'étais plutôt intéressé par l'électronique de puissance (hacheurs, redresseurs, harmoniques, etc.). Mais ça, c'était avant que le responsable de la filière Électronique et systèmes embarqués (ESE) de l'École nationale supérieure de l'électronique et de ses applications (ENSEA) ne me détrompe. Les quelques minutes qu'a durées sa présentation ont suffi à me convaincre de choisir ESE plutôt que la filière consacrée à la puissance. Depuis, je n'ai pas regretté ce choix une seule seconde. En revanche, ma perception des différentes réalités que recouvre la conception de systèmes embarqués a sensiblement évolué au fil du temps. Au cours de ma dernière année à l'ENSEA, mon intérêt se portait essentiellement sur les aspects logiciels de bas niveau.

J'ai alors fait mon stage de fin d'études au Commissariat à l'énergie atomique et aux énergies alternatives (CEA) de Saclay (91), au sein du Laboratoire de calcul embarqué, où j'ai travaillé sur le portage du logiciel système développé sur place pour une architecture multicœur (P2012) conçue en partenariat avec STMicroelectronics. Mon passage au CEA, premier contact avec le monde de la recherche, m'a ouvert les yeux sur les défis intellectuels immenses à relever et les libertés bien plus grandes offertes aux chercheurs par rapport aux ingénieurs. De plus, mon travail sur P2012 a été très enrichissant et, à l'issue de mon stage, j'étais convaincu de son potentiel, aussi bien d'un point de vue industriel qu'académique. L'idée de poursuivre en thèse est néanmoins venue tardivement en regard du calendrier institutionnel: mon stage ayant commencé en avril et les dossiers de thèse au CEA devant être déposés en mai, il était trop tard pour postuler. Vu mon intérêt pour P2012, j'ai donc cherché à intégrer STMicroelectronics, d'abord comme ingénieur, puis j'ai très vite découvert les offres de thèses CIFRE, en particulier celle qui a abouti à la rédaction du présent document. J'ai donc postulé via l'immonde interface web de ST[1]… et n'ai jamais obtenu de réponse par ce biais. En parallèle, j'ai cherché l'annonce correspondante via les canaux académiques (forum ASR, etc.) et ai contacté le laboratoire d'accueil. C'est par ce dernier biais que j'ai obtenu le premier entretien d'une longue série. Après un aller-retour à Bordeaux et un à Grenoble, un entretien scientifique, deux tests de personnalités, un entretien avec un cabinet de recrutement, un autre avec les RH de ST et enfin un dernier avec mon futur encadrant, le dossier

1 En plus d'être esthétiquement ignoble et anti-ergonomique, le site de recrutement ne fonctionne qu'avec Internet Explorer…

est envoyé à l'Agence nationale de la recherche technologique – chargée de valider les demandes de thèses CIFRE en vue de leur financement –, puis accepté deux mois plus tard. Je me dois, à ce titre, de rendre grâce à mes encadrants et à l'administration pour leur efficacité – en particulier celle de ST qui, par la suite, ne m'a pas vraiment habitué à de telles prouesses…

Voilà pour les prémisses. Il me semblait important de souligner les conditions qui m'ont accompagné jusqu'à cette thèse. Pour ce qui est du déroulement des trois années qui ont suivi, je n'énoncerai que quelques remarques à même d'éclairer la lecture du reste du document. Avant toute autre chose, il convient d'insister sur les difficultés inhérentes au travail de recherche en entreprise en général, et dans le cadre d'une thèse en particulier. D'autant plus quand les moyens mis à disposition dans ce sens sont réduits à leur plus simple expression :

- pas d'accès aux ressources documentaires[2] : Scopus, WoS, ACM, Springer, Elsevier, etc. ;
- pas de formations adaptées[3] : méthodes de recherche et outils bibliographiques, rédaction, publication, etc. ;
- pas de façon simple de contacter les autres doctorants ;
- restrictions inacceptables dans l'utilisation des moyens informatiques :
  - toutes les connexions vers l'extérieur sont bloquées par défaut, en particulier : accès web uniquement via proxy authentifié et *man-in-the-middle* en SSL, SSH interdit sauf dérogation longue et difficile à obtenir puis après authentification durant une courte fenêtre temporelle, etc. ;
  - pas de PC de bureau sous Linux, sauf dérogation motivée et très difficile à obtenir, puis, le cas échéant, pas de wifi, pas de VPN, image installée inadaptée au matériel, pas d'accès *root*, etc.
  - sous Windows, interdiction de gérer ses courriels avec autre chose que M$ Outlook.

Par ailleurs, l'environnement de travail est celui d'un ingénieur, pas celui d'un chercheur. Les discussions sont donc plus centrées sur la satisfaction du client que sur les défis scientifiques et technologiques à relever. En revanche, les ressources propres d'une grande entreprise, bien supérieures à celles d'un laboratoire de recherche académique, lui permettent de mettre en œuvre des moyens sans commune mesure, notamment pour ce qui est de l'informatique :

- ressources de calcul parfaitement dimensionnées, aussi bien en nombre qu'en puissance : il n'y a jamais d'attente pour le lancement d'une simulation ;
- taux de plantage et de panne très faibles ;
- stockage de toutes les données sur disques redondés avec accès extrêmement rapide ;
- serveurs de travail à très haute capacité mémorielle, y compris pour les applications graphiques.

2 Sauf IEEEXplore, à condition de passer par un portail fort peu pratique…

3 Pour être tout à fait exact, en tout et pour tout, en trois ans, une seule formation à destination des doctorants a été proposée par ST, sur le thème de la propriété intellectuelle et des brevets, vantant les mérites de ces derniers. Sujet primordial pour mener à bien un travail de recherche…

Un autre avantage supposé du travail en entreprise est l'existence de débouchés sur le marché qui sont censés garantir – ou, pour le moins, favoriser – la réutilisation des produits de la recherche en dehors du cercle académique. Dans mon cas, cet espoir de voir les fruits de son travail largement diffusés, même sous forme commerciale, n'a été qu'un mirage. En effet, le projet P2012 (devenu entre-temps STHORM) a été abandonné par ST en plein milieu de ma thèse. Les autres personnes travaillant dessus ont été transférées dans diverses équipes ou ont quitté l'entreprise. En conséquence, le modèle de programmation servant de base à mes travaux (PEDF) n'a dès lors plus été maintenu et le support est devenu pour ainsi dire inexistant, rendant d'autant plus difficiles les développements afférents.

Outre les difficultés liées au travail en entreprise, un autre inconvénient – certes moindre – a marqué le déroulement de cette thèse, côté Inria cette fois-ci : j'étais la seule personne du laboratoire dont le domaine de recherche était l'électronique embarquée, tous les autres travaillant sur le parallélisme et le calcul intensif. Côtoyer des chercheurs en informatique a néanmoins eu l'avantage de me faire prendre conscience des différences nombreuses et fondamentales entre ces deux domaines que je croyais initialement bien plus proches ; j'y consacre d'ailleurs une part substantielle du premier chapitre. Cela dit, on aurait tort de sous-estimer l'impact négatif du déracinement, qu'il soit de l'ordre du domaine – faire de l'électronique dans un laboratoire d'informatique – ou de l'activité – être chercheur au milieu d'ingénieurs. En effet, le travailleur déraciné souffre d'un certain sentiment d'exclusion résultant de son incapacité à prendre pleinement part à la vie professionnelle du milieu dans lequel il baigne sans s'y fondre. Par ailleurs, le fait qu'il ne parle pas le même langage et ne raisonne pas de la même manière que ses collègues gêne la communication.

Enfin, d'un point de vue pratique – et contrairement à ce que l'on pourrait imaginer au premier abord –, partager son temps entre deux lieux de travail à 600 km l'un de l'autre ne relève pas forcément du défi logistique. Dans mon cas, le laboratoire remboursant le logement à Bordeaux, je pouvais garder deux domiciles pendant une partie de l'année, ce qui facilitait les échanges. En fin de compte, le surcoût financier a pour moi été quasi nul étant donné la prise en charge des déplacements et déménagements sous forme de missions, bien que le paiement en fût différé.

Pour conclure, malgré les nombreuses difficultés rencontrées, cette thèse a été pour moi une expérience globalement fort positive à la fois aux points de vue professionnel et personnel. Mon envie de faire de la recherche est venue tardivement mais se trouve maintenant bien ancrée et adossée à celle d'enseigner[4]. C'est pourquoi, après un post-doc au CEA sur la gestion d'architectures hétérogènes embarquées, je me destine aux fonctions d'enseignant-chercheur.

4 Bien que n'étant pas doctorant contractuel, j'ai eu la chance de pouvoir m'initier à la pédagogie en suivant des formations et en encadrant des TD et TP de réseaux à l'Université de Bordeaux.

Table des matières

Introduction

Plus que jamais, l'électronique multimédia imprègne nos vies. Et cette tendance ne semble pas près de se démentir. Après que les supports analogiques tels que les cassettes eurent laissé la place au numérique avec les CD, ces derniers furent à leur tour supplantés par le MP3. À chaque nouveau format son lecteur portable : en trente ans, l'évolution des baladeurs a été fulgurante [1]. Alors que les premiers Walkman de Sony étaient lourds, volumineux et énergivores, l'iPod d'Apple[5] est fin, léger et capable de distiller des heures de musique sans être rechargé. Mais cette évolution ne s'arrête pas à la musique : à partir de 2005, la lecture de vidéos fait immersion dans un monde où l'audio régnait jusqu'alors en maitre. Le marketing remplissant son office, ce qui passait quelque temps auparavant pour une idée farfelue devient rapidement un passe-temps indispensable. Ainsi l'expansion des appareils capables de lire des vidéos ne connaît-elle pas de limites. Les fabricants de téléphones mobiles ne tardent pas à surfer eux aussi sur la vague multimédia, complétant la panoplie des usages n'étant pas directement liés à la téléphonie. En fin de compte, l'essor de terminaux embarqués de plus en plus diversifiés disposant de fonctionnalités liées au son et à l'image équivalentes voire supérieures à celles des appareils dédiés a profondément modifié les pratiques des utilisateurs, au point de poser des problèmes sanitaires [2].

Ces mutations peuvent s'expliquer par les avancées technologiques réalisées dans différents domaines : les télécommunications avec les réseaux sans fil à haut débit, l'électronique embarquée avec une nouvelle génération d'architectures spécialisées, et la compression vidéo. Leur convergence se traduit, par exemple, par la possibilité d'assister en direct depuis son téléphone portable à des retransmissions filmées d'évènements sportifs ou culturels. Si cet état de fait peut laisser penser que l'opération consistant à décoder un flux d'images serait devenue une tâche anodine à la portée de n'importe quel système embarqué, il n'en est rien. En réalité, la complexité des applications précède bien souvent celle du matériel ; de ce fait, il revient aux concepteurs de ce dernier de suivre le rythme des innovations. Or, la conception des systèmes embarqués est un processus long et qui n'admet pas les imperfections tolérées dans le développement des logiciels informatiques. Des solutions doivent donc être proposées en amont, avec la caractéristique de pouvoir s'adapter rapidement aux nouvelles exigences. Il s'agit généralement d'un assortiment d'unités de calcul programmables et d'accélérateurs matériels regroupés dans un même système sur puce multiprocesseur (*multiprocessor system on chip*, MPSoC). Le très populaire Raspberry Pi[6], bien que s'adressant à un public averti, est représentatif de cette tendance : il comprend un cœur ARM destiné à exécuter du code logiciel généraliste, et un processeur graphique Broadcom permettant de décoder des vidéos de qualité équivalente au format Blu-ray. De tels systèmes sont qualifiés

5 Pour ne citer que les deux plus grands succès commerciaux.
6 http://www.raspberrypi.org/

d'hétérogènes au sens où ils comprennent plusieurs types de processeurs – deux dans le cas du Raspberry Pi.

Du point de vue des applications, la complexité se traduit par une implantation difficile sur le matériel existant. C'est le cas, en particulier, des applications à flux de données (*streaming*) dont les algorithmes de décodage vidéo utilisés pour le visionnage sur les terminaux des utilisateurs sont de bons représentants. Pour cette classe d'applications, l'implantation peut être facilitée par le recours à des modèles à flot de données (*dataflow*). Côté matériel, il est également possible de favoriser l'exécution en adaptant la plateforme à ce modèle, de sorte qu'il puisse cohabiter avec le paradigme Von Neumann classique : on parle alors d'architecture hybride, ce qui constitue un cas particulier d'hétérogénéité. À cela s'ajoute le caractère dynamique et imprévisible de ces applications, dû à la complexité des algorithmes récents, qui pèse sur l'efficacité de l'exécution et soulève d'importantes questions d'ordonnancement auxquelles s'attache à répondre cette thèse.

Cadre de la thèse et contributions

STMicroelectronics a commencé à travailler en 2009 sur un projet de plateforme multicœur hybride embarquée répondant à ces critères. L'absence d'outils existants satisfaisants pour le placement efficace du code applicatif sur une telle architecture a conduit l'entreprise à faire appel au savoir-faire de l'équipe Runtime d'Inria dans ce domaine. Cette thèse CIFRE s'inscrit dans ce contexte.

En vue de réduire le fossé entre les applications et l'architecture, il est nécessaire de recourir à des modèles de calcul, de programmation et d'exécution adaptés. Le dynamisme et la complexité d'une part, l'hybridité d'autre part, rendent ce besoin encore plus criant. C'est pourquoi le premier objectif de la thèse est de tenter de combler ce fossé en proposant un modèle d'exécution à flot de données adapté à ces caractéristiques particulières. Cette solution se différencie de l'existant par le support d'un nouveau type de paramètres à même de capturer les facteurs de variabilité de l'application.

L'exécution efficace des applications étant étroitement liée à l'ordonnancement, cet aspect est également abordé. La question cruciale du choix entre approche statique et dynamique est soulevée. Il semblerait que la seconde soit plus adaptée pour les applications à forte variabilité : cette conjecture est discutée et des outils d'évaluation sont présentés. Par ailleurs, le problème fondamental des contraintes mémorielles dans les systèmes embarqués est également traité, et une solution concrète fondée sur des techniques existantes et éprouvées est apportée. Enfin, le modèle d'exécution proposé est lui aussi adossé à un ordonnanceur spécifique dont la stratégie peut être définie par l'utilisateur pour plus de flexibilité.

Organisation du document

La présente thèse s'articule autour de six chapitres. Le premier expose le contexte et les hypothèses de travail, et formule la problématique. Le second décrit le modèle d'exécution à flot de données proposé, en quoi il se distingue de l'existant et les réponses qu'il apporte aux problèmes soulevés. Le troisième aborde la question de l'ordonnancement sous contrainte de mémoire et propose un ajustement d'une classe répandue d'heuristiques en vue d'y introduire des garanties fortes. Le quatrième revient brièvement sur des considérations relatives à l'implémentation et au

protocole expérimental adoptés pour le modèle présenté au chapitre 2. Le cinquième relate les contributions apportées au logiciel système en vue de supporter le modèle défini précédemment. Enfin, le dernier chapitre conclut l'exposé en présentant les résultats expérimentaux liés à l'évaluation des travaux présentés dans les chapitres 2 et 5, à savoir le modèle d'exécution adossé au support logiciel.

Chapitre 1. Contexte et environnement

Ce chapitre a pour objet de décrire le contexte scientifique et technique de cette thèse. La première section évoque les applications visées en insistant sur leurs particularités et les défis qu'elles soulèvent. La deuxième section explicite les spécificités de l'électronique embarquée et notamment en quoi celles-ci doivent impérativement être prises en compte tout au long du processus de développement. La troisième section est consacrée au modèle d'architecture ciblé ainsi qu'à la plateforme réelle sur laquelle ont eu lieu les expérimentations de la thèse. Enfin, la dernière section expose formellement le problème abordé.

1.1 Applications: traitement d'images et décodage vidéo

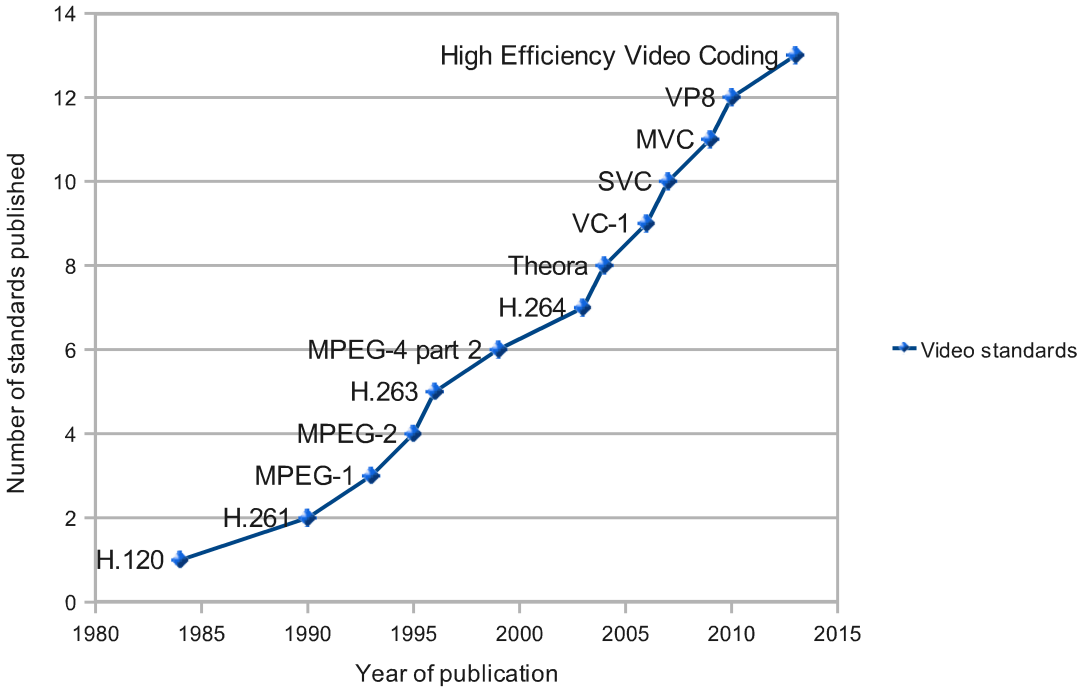


FIGURE 1.1 – *Chronologie de la publication des normes de compression vidéo. Source: Wipliez [80].*

À l'heure où la convergence des terminaux numériques repousse les limites de l'intégration de fonctionnalités multimédias autrefois dévolues à des appareils *ad hoc*, il n'est pas rare de rencontrer des téléphones mobiles capables de reproduire des contenus vidéo en diffusion continue émanant d'un réseau sans fil. De fait, l'électronique grand public du divertissement – baladeurs, tablettes, décodeurs télévisuels, etc. – a connu une croissance exponentielle durant la dernière décennie, et qui, selon toute vraisemblance, ne devrait pas se démentir au cours des prochaines années, en particulier dans le domaine du décodage vidéo. En effet, les normes de compression

audiovisuelle ont connu dans la même période une évolution effrénée (voir figure 1.1) qui a permis d'atteindre une qualité de restitution sans précédent. Néanmoins, cette amélioration ne va pas sans un accroissement de la complexité [3] des algorithmes employés, qui rend l'implémentation d'autant plus ardue (cf. section 1.1.2), posant de nombreux défis.

Pour cette raison, cette thèse s'intéresse prioritairement au décodage vidéo, mais également à toute une classe d'applications dites « à flux de données » – c'est-à-dire dont la structure et le fonctionnement sont tels que, d'un point de vue haut niveau, les données « s'écoulent » entre unités fonctionnelles en cascade – dont la plupart des algorithmes dédiés au traitement du signal sont des exemples éloquents. Les travaux présentés dans les prochains chapitres sont donc applicables à l'ensemble de cette classe, et en particulier à un sous-ensemble rendu difficile à appréhender du fait d'un fort dynamisme caractérisé par une imprédictibilité et une variabilité nécessitant des reconfigurations à l'exécution.

1.1.1 Généralités sur le codage vidéo

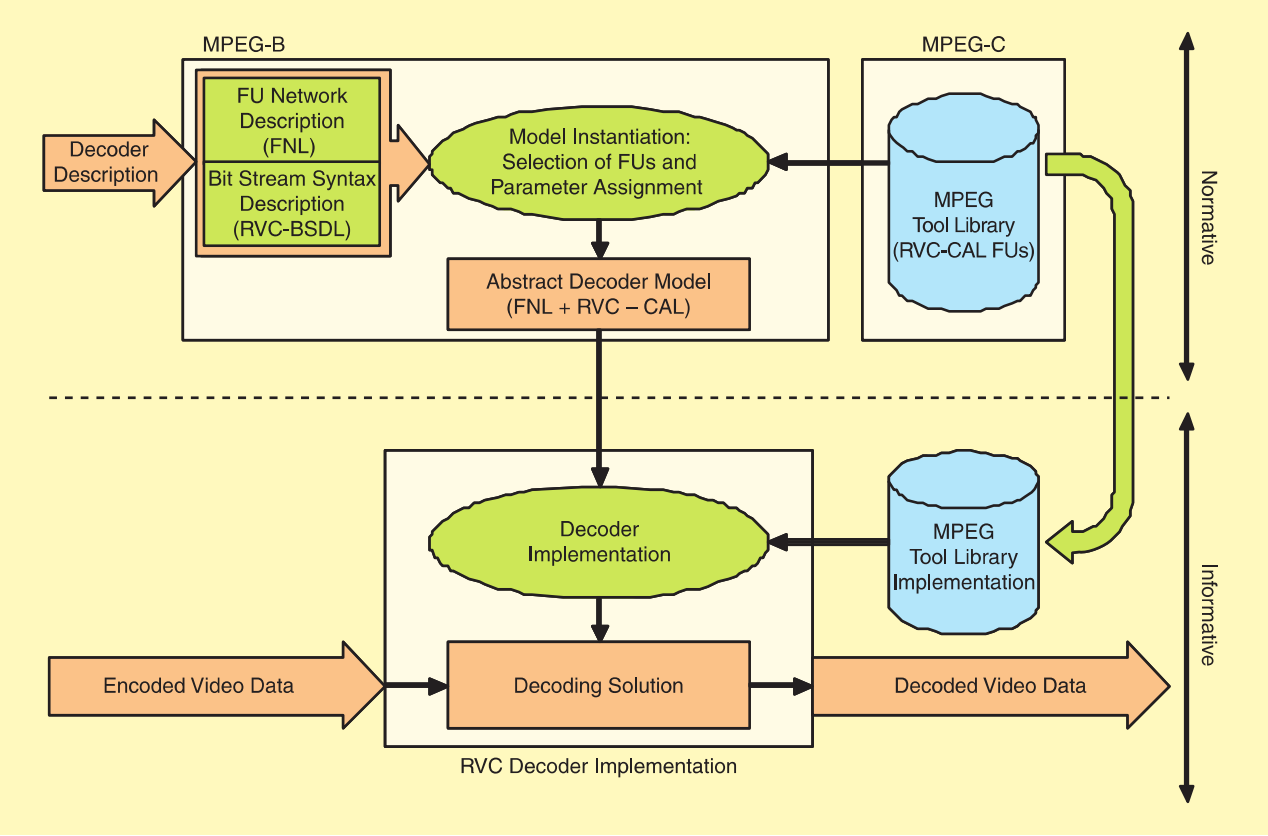


FIGURE 1.2 – *Vue d'ensemble de la norme RVC. Source: Mattavelli et coll. [81].*

Les évolutions récentes des algorithmes d'encodage et décodage (codecs) vidéo visant à améliorer l'expérience des utilisateurs – image ultra haute définition (UHD, 4K voire 8K), 3D, etc. – ainsi que l'efficacité accrue de la compression justifient un besoin croissant en puissance de calcul. Parmi les codecs modernes les plus répandus, on peut citer, par ordre chronologique [4] :

- MPEG-2 : DVD, SVCD, télévision numérique terrestre (TNT) simple définition ;
- H.264 (ou MPEG-4 partie 10/AVC) : HD-DVD, Blu-ray, TNT haute définition ;
- HEVC (ou H.265) : télévision UHD (4K, 8K).

Devant la complexité croissante des codecs et l'apparition de plus en plus fréquente de nouveaux venus, une nouvelle norme de codage vidéo reconfigurable (RVC) [5] a été introduite pour adapter leur description de façon à faciliter leur implémentation sur des cibles diverses, aussi bien matérielles que logicielles. Comme le montre la figure 1.2, cette norme comprend une restriction du langage CAL [6] (RVC-CAL) permettant de décrire l'algorithme sous forme d'un modèle de décodeur abstrait (*abstract decoder model*) qui remplace les implémentations de référence en C ; elle fournit également une bibliothèque standard (MPEG *Tool Library*) comprenant des briques de base (*functional units*, FUs) pour construire de tels décodeurs, mais laisse la porte ouverte à des implémentations propriétaires. Le chapitre 4 présente Orcc, une chaine d'outils dédiée à RVC.

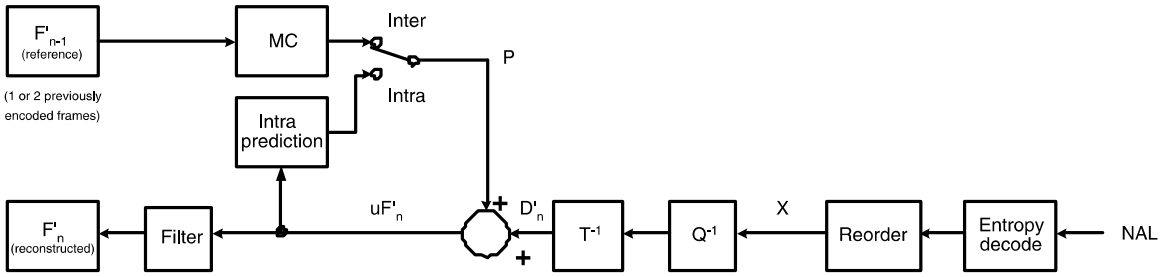


FIGURE 1.3 – *Décodeur H.264. Source : Richardson [8].*

Cette approche est d'autant plus valorisable que tous les codecs sont fondés sur les mêmes bases et partagent une structure commune. Une séquence vidéo est composée d'images ou *trames*, divisées en groupes de pixels – typiquement de forme carrée et de taille 16 ×16 [7] –, appelés *macroblocs*, regroupés en séquences contigües dans l'ordre de balayage[7], appelées *tranches*. La figure 1.3 représente le schéma d'un décodeur H.264, mais le principe reste le même pour les autres codecs. La première partie (à droite sur la figure) consiste à décoder les échantillons, appelés résidus, à les réordonner, et à leur appliquer une quantification et une transformée en cosinus discrète inverse (IDCT). La deuxième (à gauche sur la figure) comprend la prédiction intra- ou inter-images, ainsi que le filtrage[8].

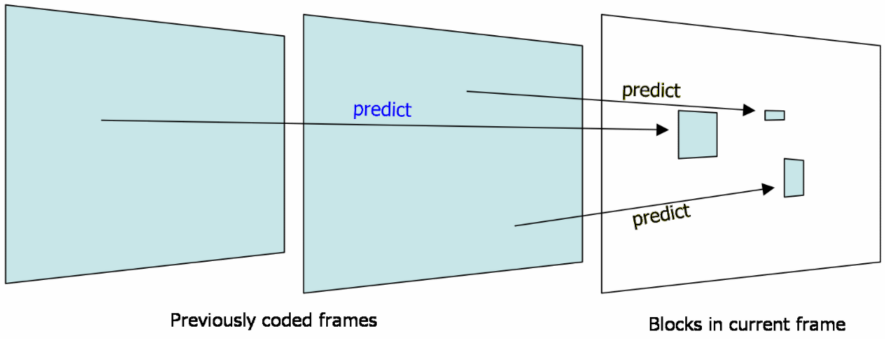


FIGURE 1.4 – *Inter-prédiction. Source : Richardson [82].*

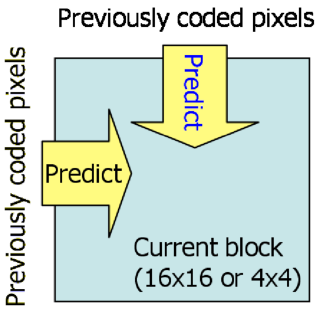


FIGURE 1.5 – *Intra-prédiction. Source : ibid.*

L'intra-prédiction [9] (fig. 1.4) est une méthode de codage qui réduit les redondances spatiales en utilisant les pixels limitrophes déjà codés (à gauche et au-dessus) au sein d'une même image pour

7 Historiquement, les écrans analogiques affichaient les images par projection d'un faisceau qui balayait la surface horizontalement de gauche à droite, ligne par ligne de haut en bas. Les codecs numériques ont conservé cet ordre pour le traitement des macroblocs.

8 Se reporter à Richardson [8] pour les détails de chaque étape.

reconstruire un macrobloc. L'inter-prédiction [10] (fig. 1.5) opère quant à elle dans le domaine temporel, en s'appuyant sur des échantillons provenant d'images déjà décodées, par le biais d'un mécanisme appelé compensation de mouvement et connu pour être le principal goulet d'étranglement dans le décodage vidéo en temps réel [11]. L'alternance entre macroblocs intra- et inter-codés au sein d'une même trame est un bon exemple de variabilité induisant un fort dynamisme chez tous les codecs vidéo récents.

1.1.2 Questions d'implémentation

Le processus de décodage vidéo est d'autant plus délicat qu'il doit le plus souvent être effectué en temps réel: le flux brut est traité au fur et à mesure qu'il est lu, et si une trame n'a pu être correctement reconstruite au terme du temps imparti, l'expérience de l'utilisateur risque d'être dégradée par un affichage saccadé, des artéfacts dans l'image, etc. À titre d'exemple, un décodeur H.264 certifié 1080p[9] doit être en mesure de décoder jusqu'à 589 824 macroblocs par seconde, ce qui, avec un seul processeur cadencé à 600 MHz, correspond à un budget de 1 017 cycles par macrobloc. Dans ces conditions, il est clair qu'un traitement séquentiel par un processeur embarqué généraliste n'est pas envisageable.

Pourtant, jusqu'à il y a peu, les pratiques courantes pour l'implémentation de décodeurs vidéo étaient essentiellement centrées sur des solutions logicielles avec peu – voire pas – de parallélisme. En effet, la conception matérielle est longue et coûteuse, ce qui est incompatible avec les impératifs de réduction des temps de mise sur le marché dus à la concurrence industrielle féroce. En revanche, le développement logiciel est un processus beaucoup plus rapide, itératif et flexible, qui permet notamment la correction de défauts de conception à posteriori. De plus, les implémentations de référence des décodeurs étant, jusqu'à l'avènement récent de RVC, purement séquentielles, la parallélisation était une tâche fastidieuse. Mais aujourd'hui, la complexité des derniers codecs est telle que la seule solution viable consiste à avoir recours à un parallélisme fonctionnel, d'une part, et à des accélérateurs matériels dont chacun est spécialisé dans l'exécution d'une tâche bien déterminée, d'autre part.

Cette solution hybride, alliant la flexibilité du logiciel et l'efficacité – à la fois en termes de rapidité d'exécution, de surface de silicium et de consommation énergétique – du matériel, peut prendre la forme d'une plateforme hétérogène modulable extensible telle que celle décrite à la section 1.3.2. La mise en œuvre d'une telle solution implique en outre de disposer d'une description de l'application à haut niveau d'abstraction permettant de cibler à la fois le matériel et le logiciel: les modèles à flot de données présentés au chapitre 2 répondent à cette nécessité.

1.2 Spécificités de l'embarqué

Cette section tente d'abord de circonscrire ce que l'on entend usuellement par le terme « embarqué », employé dans des contextes très divers, avant de définir aussi précisément que possible l'acception retenue dans le cadre de cette thèse. Ses spécificités seront ensuite étudiées, à la fois sur les plans matériel et logiciel.

9 La notation 1080p signifie 1080 lignes de pixels par image et balayage progressif. Les chiffres fournis ici correspondent à un décodeur de niveau 5.

1.2.1 Différences entre électronique et informatique

Samek [12] définit un système embarqué comme « ayant un ordinateur enfoui en son sein mais n'étant pas perçu comme tel par l'utilisateur », ce qui n'est pas dénué de subjectivité. Une autre définition parfois adoptée par les enseignants en guise d'introduction est qu'il s'agit d'un système informatique qui n'est pas un ordinateur. Ces deux définitions soulignent, à juste titre, l'aspect informatique, mais omettent – peut-être volontairement – le non moins important aspect électronique. Celle donnée par Wikipédia[10] comble ce manque: « Un système embarqué (ou système enfoui) est défini comme un système électronique et informatique autonome, souvent temps réel, spécialisé dans une tâche bien précise. » Est-ce à dire que les deux composantes coexistent toujours à parts égales dans de tels systèmes? L'observation de l'extrême hétérogénéité qui caractérise la vaste classe de l'embarqué permet d'y répondre: la dimension informatique est par exemple bien plus prépondérante dans un téléphone dit « intelligent » que dans un commutateur de réseau. Il apparaît donc nécessaire d'opérer une distinction entre électronique embarquée et informatique embarquée. Hennesy & Patterson [13] choisissent comme critère le support de logiciels tiers, ce qui implique la présence d'un système d'exploitation. Au contraire, en électronique embarquée, seul un logiciel système est nécessaire pour gérer un ensemble prédéfini d'applications. Le tableau 1 résume les caractéristiques des différentes classes de systèmes.

	Perçu par l'utilisateur comme un ordinateur	Système d'exploitation	Exemple
Électronique embarquée	Non	Non	Décodeur TV
Informatique embarquée	Pas forcément	Oui	Raspberry Pi
Informatique généraliste	Oui	Oui	PC de bureau

TABLEAU 1 – *Caractéristiques des différentes classes de systèmes.*

De manière générale, l'objectif principal de conception en électronique embarquée est de minimiser les coûts de production (surface de silicium) tout en garantissant le niveau de performance requis (prédictibilité), et non d'atteindre la performance maximale. De ce fait, les métriques utilisées sont la densité de performance (en millions d'instructions par seconde par millimètre carré [MIPS/mm²]) et la densité de puissance (en watts par millimètre carré [W/mm²]), plutôt que la puissance brute en MIPS ou en nombre d'opérations flottantes par seconde (FLOPS). Enfin, comme le soulignent Sheng et coll. [14], le modèle commercial est également différent: le haut niveau de désagrégation de l'industrie électronique, où les fabricants sont souvent spécialisés dans des domaines précis, favorise une approche horizontale – plusieurs processeurs de différents fabricants dans un même système – plutôt que verticale – chaque fournisseur contrôle l'ensemble de la chaine du matériel au logiciel. S'ajoute à cela un cycle de vie raccourci pour l'embarqué: la pression visant à la réduction du temps de mise sur le marché impose de fréquentes évolutions sur les plateformes produites, avec pour conséquence la nécessaire adaptabilité de la pile logicielle associée.

10 http://fr.wikipedia.org/wiki/Syst%C3%A8me_embarqu%C3%A9

1.2.2 Matériel

Les systèmes embarqués étant définis comme « spécialisés dans une tâche bien précise[11] », il est clair que les architectures et micro-architectures généralistes que l'on trouve dans les ordinateurs sont hautement inadaptées à ce dessein. Au contraire, la spécialisation de l'architecture est un facteur décisif qui permet d'atteindre le niveau de performance souhaité sans gaspillage de ressources. Des exemples de telles architectures sont données à la section 1.3.3.

Parmi les spécificités du matériel embarqué, on peut citer la très haute intégration des systèmes sur puces (*systems on chips*, SoC) pour notamment satisfaire aux contraintes d'espace et de mobilité, mais aussi le calcul sur composants discrets de toutes sortes: processeurs généralistes ou spécialisés (par exemple dans le traitement du signal), circuits logiques programmables (p. ex. FPGA ou CPLD), accélérateurs matériels, etc. Les contraintes d'espace et de mobilité imposent également des limites sur la quantité et la nature de la mémoire: les disques durs, lourds et volumineux, sont délaissés au profit de mémoire de type Flash pour le stockage de masse, et la mémoire volatile statique est largement utilisée, pas uniquement pour les caches (voir l'exemple de STHORM à la section 1.3.2). Les processeurs sont aussi plus simples: le Sandy Bridge d'Intel dispose de dix-sept étages de pipeline, tandis qu'un STxP70 de STMicroelectronics n'en a que cinq. D'autre part, l'adressage est direct afin d'éviter le surcoût lié à la mémoire virtuelle: pas de matériel dédié à sa gestion – mais éventuellement une unité de protection visant à prévenir les accès illicites – ni de couche logicielle supplémentaire dans le noyau; en pratique, le chargement de l'application s'en trouve simplifié, l'imprédictibilité réduite et les performances améliorées.

Bien souvent, la spécialisation de la micro-architecture va de pair avec un jeu d'instructions réduit et simplifié, permettant une taille de code diminuée et un décodage plus aisé. Une fonctionnalité particulièrement intéressante, et qui fait pourtant défaut à la plupart des architectures généralistes, est le support des évènements: ils permettent à plusieurs processeurs de communiquer de façon moins intrusive et beaucoup plus efficace que par le biais d'interruptions. Enfin, tous les systèmes embarqués sont conçus avec le souci de minimiser la consommation énergétique, à plus forte raison pour les appareils mobiles dont l'autonomie doit être préservée, ainsi que la dissipation thermique.

1.2.3 Logiciel

Outre les différences matérielles, les systèmes embarqués se distinguent également par leur programmation qui, selon Samek [12], nécessite une approche fondamentalement différente. En effet, le développement pour les cibles embarquées offre à la fois des facilités – par exemple des hypothèses fortes sur l'environnement logiciel et matériel – et des difficultés – comme la gestion manuelle de la mémoire – totalement étrangères à l'informatique générique. Le développeur embarqué dispose d'un contrôle accru – accès direct aux périphériques via des registres mappés en mémoire – qui peut être vu comme un avantage ou un inconvénient: les garde-fous mis en place par le système d'exploitation sont autant d'entraves, tandis que les primitives de haut niveau exposées ajoutent de nombreux niveaux d'indirection pouvant être jugés inutiles voire néfastes. Au contraire du développement informatique conventionnel où le recours à de larges bibliothèques tierces est la norme, l'embarqué adopte une approche minimaliste privilégiant la simplici-

11 Le niveau de précision de la tâche en question peut ou non impliquer la Turing-complétude, selon les cas.

té au détriment de la réutilisabilité; à plus forte raison pour des systèmes critiques, un développeur ne prendra pas le risque d'introduire du code qu'il ne maîtrise pas parfaitement. Il est également des compromis jugés inhabituels: les contraintes de mémoire sont parfois telles qu'elles conduisent à préférer réduire l'empreinte du code quitte à perdre en performance.

1.2.3.1 Inadéquation des techniques de développement généralistes

Cette sous-section énumère un certain nombre de techniques largement répandues en informatique, à tel point qu'il ne viendrait à l'esprit d'aucun développeur conventionnel de les remettre en cause; et pourtant, ce qui suit expose les raisons pour lesquelles elles doivent, autant que faire se peut, être évitées dans un contexte embarqué.

2.3.1.1 Allocation de mémoire dynamique

Comme le souligne Samek [12], allouer de la mémoire dynamiquement fait appel à une structure de donnée de type tas qui, à elle seule, concentre un grand nombre de problème. En premier lieu, le tas gaspille une quantité de mémoire certes négligeable dans un environnement de type PC mais rédhibitoire dans le cas d'un système embarqué: cela est dû pour partie au stockage des méta-données, mais surtout à la fragmentation qui se forme au fur et à mesure que des segments de tailles diverses sont alloués puis libérés, à tel point qu'il est possible que la mémoire restante soit suffisante pour une allocation mais que celle-ci ne puisse pas avoir lieu par défaut de contigüité. De plus, il est extrêmement difficile de prédire la taille adéquate du tas, ce qui amène généralement à le surdimensionner grossièrement. Par ailleurs, les primitives d'allocation et de libération sont coûteuses, non déterministes – ce qui est inacceptable pour un grand nombre d'applications subissant des contraintes d'exécution en temps réel dur – et non réentrantes.

En outre, les dangers de l'allocation dynamique bien connus de tous les développeurs, sont encore plus prégnants en électronique embarquée: une seule fuite, même de taille réduite, peut suffire à saturer la mémoire; la technique consistant à sécuriser un pointeur en le mettant à zéro ne fonctionne pas si l'adresse nulle existe – c'est le cas de STHORM; le débogage est difficile car l'on ne sait pas à l'avance où seront stockées les données et, en l'absence de mémoire virtuelle, les erreurs de segmentation n'existent pas.

2.3.1.2 Édition des liens et chargement dynamiques

L'éditeur de liens est un programme qui modifie le code objet d'une application en remplaçant les références sous forme de symboles par leurs adresses relatives, tandis que le chargeur copie l'application de la mémoire de stockage vers la mémoire exécutable [15]. Les deux étapes les plus longues et complexes [16] sont la résolution des symboles et la relocalisation, qui sont, pour cette raison, généralement effectuées juste après la compilation: on parle alors d'édition des liens statique; dans le cas dynamique, au contraire, ces deux étapes ont lieu à l'exécution. Autant, sur un PC, cette dernière méthode peut être quasi transparente, autant, en électronique embarquée, son coût est la plupart du temps rédhibitoire.

De manière générale, les inconvénients de l'édition des liens – et, dans une moindre mesure, du chargement – dynamique sont nombreux. On peut en citer quelques-uns:

- ce qui pourrait être fait une seule fois à la compilation est répété inutilement à chaque exécution pour produire toujours le même résultat, et, la plupart du temps, de manière bien moins efficace, ralentissant d'autant le chargement de l'application ;
- les erreurs liées à des références non résolues ou des symboles introuvables sont détectées et signalées tardivement ;
- le surcoût est proportionnel au nombre de relocalisations et de symboles à chercher [16], ainsi qu'à la taille du programme, ce qui incite le développeur à réduire le nombre de références externes, quitte à dégrader la qualité du code ;
- au cours de l'exécution, chaque accès à un membre (objet ou fonction) d'une bibliothèque dynamique ajoute au moins un niveau d'indirection et consomme inutilement des cycles de processeur.

1.2.3.2 Simulation et débogage

Contrairement au monde de l'informatique où la plupart des expérimentations ont lieu sur du matériel réel, en électronique embarquée les prototypes sont souvent indisponibles voire inexistants. En effet, le temps et le coût de leur fabrication sont élevés et ne peuvent être assumés qu'à un stade avancé du cycle de développement. Le recours à des plateformes de simulation s'impose donc (cf. chapitre 4), ce qui conduit à des différences d'approche substantielles. Au chapitre des avantages, on peut noter la possibilité d'instrumenter le simulateur de façon à mener des mesures non intrusives, les facilités d'introspection permises par les modèles de plus haut niveau, ainsi que la parallélisation des expériences. En revanche, les inconvénients sont nombreux et prennent parfois le pas sur les bénéfices :

- temps de simulation d'autant plus longs que la modélisation est fine (jusqu'à un million de fois plus lent que le matériel), une conséquence non négligeable étant la nécessité de compiler le logiciel à un niveau d'optimisation élevé qui rend le débogage en assembleur obligatoire et laborieux ;
- source de bogues supplémentaire quand le code n'est pas le même que celui de référence (utilisé pour la synthèse) ;
- mesures dépendantes de la précision du modèle retenu (cf. chapitre 4), donc parfois très approximatives voire impossibles.

En outre, le recours massif aux sorties formatées est à éviter – car très coûteux en temps et en mémoire, intrusif et nécessitant des recompilations fréquentes, elles-mêmes très longues du fait de la spécificité des générateurs de code et de la complexité des optimisations – quand il n'est pas tout simplement impossible. En effet, pour ce faire, une interface de sortie universelle est nécessaire, et bon nombre de systèmes embarqués n'en disposent pas.

1.3 Plateforme cible

Cette section décrit de façon générale le type d'architecture visé, puis expose plus en détails les spécificités de la plateforme STHORM, cible des travaux de cette thèse, et enfin conclut en présentant des architectures similaires.

1.3.1 Description du modèle d'architecture

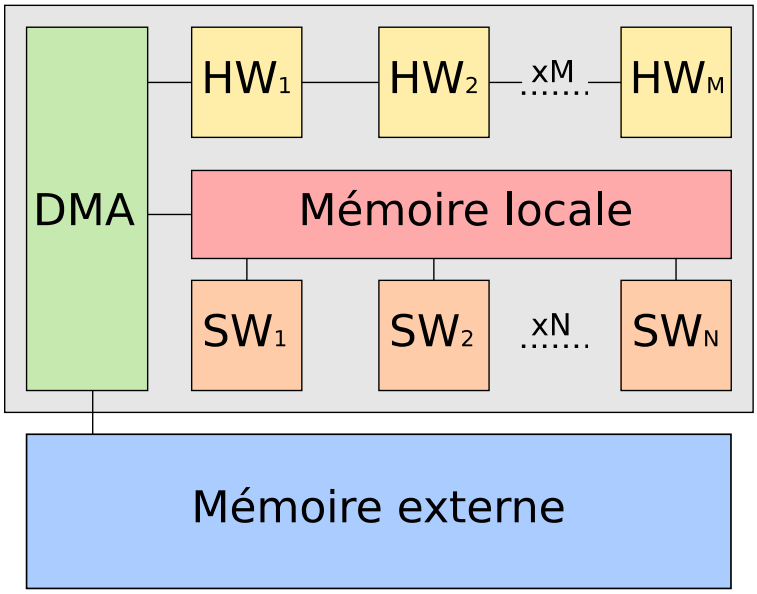


FIGURE 1.6 – *Modèle de plateforme.*

Le modèle d'architecture représenté par la figure 1.6 est une plateforme hétérogène comprenant à la fois des processeurs généralistes programmables (*software processing elements*, SWPE), peu performants mais offrant l'avantage de la flexibilité, et des accélérateurs spécialisés câblés (*hardware processing elements*, HWPE), très efficaces mais conçus pour une tâche unique. Les unités de calcul, quel que soit leur type, sont totalement interconnectées via un réseau sur puce. Elle dispose d'une mémoire hiérarchique à deux niveaux : une mémoire locale partagée en quantité limitée mais à accès rapide et une mémoire externe globale à capacité élevée mais forte latence. L'implémentation de la première se fait typiquement avec de la mémoire statique, offrant des performances similaires à celles d'un cache, et ayant de surcroît l'avantage d'une plus faible consommation énergétique que la mémoire dynamique généralement utilisée pour la seconde. Un contrôleur d'accès direct à la mémoire (DMA) est présent et permet de soulager les processeurs des tâches de communication entre les différents niveaux de la hiérarchie.

1.3.2 Présentation de STHORM

La plateforme multicœur hybride à basse consommation de STMicroelectronics (*ST Hybrid lOw poweR Manycore*, STHORM) a pour but d'accélérer des applications nécessitant un nombre d'opérations par seconde allant de quelques centaines de millions à plusieurs centaines de milliards, typiquement dans les domaines du traitement d'images, du décodage vidéo, de la réalité augmentée et de la compréhension de l'environnement.

Comme l'illustre la figure 1.7, STHORM est constitué d'une fabrique divisée en groupes de calcul (*clusters*), d'un contrôleur (*fabric controller*) et d'un réseau sur puce asynchrone assurant l'interconnexion de l'ensemble. Les groupes de calcul comprennent jusqu'à dix-sept processeurs chacun – avec des flux d'instructions indépendants, et partageant une mémoire de premier niveau accessible en un cycle –, jusqu'à deux contrôleurs DMA et un support matériel pour la synchronisation et l'accélération des tâches d'ordonnancement. La plateforme supporte jusqu'à trente-deux groupes de calcul, mais, dans le cadre de cette thèse, seule une configuration mono-groupe a été envisagée.

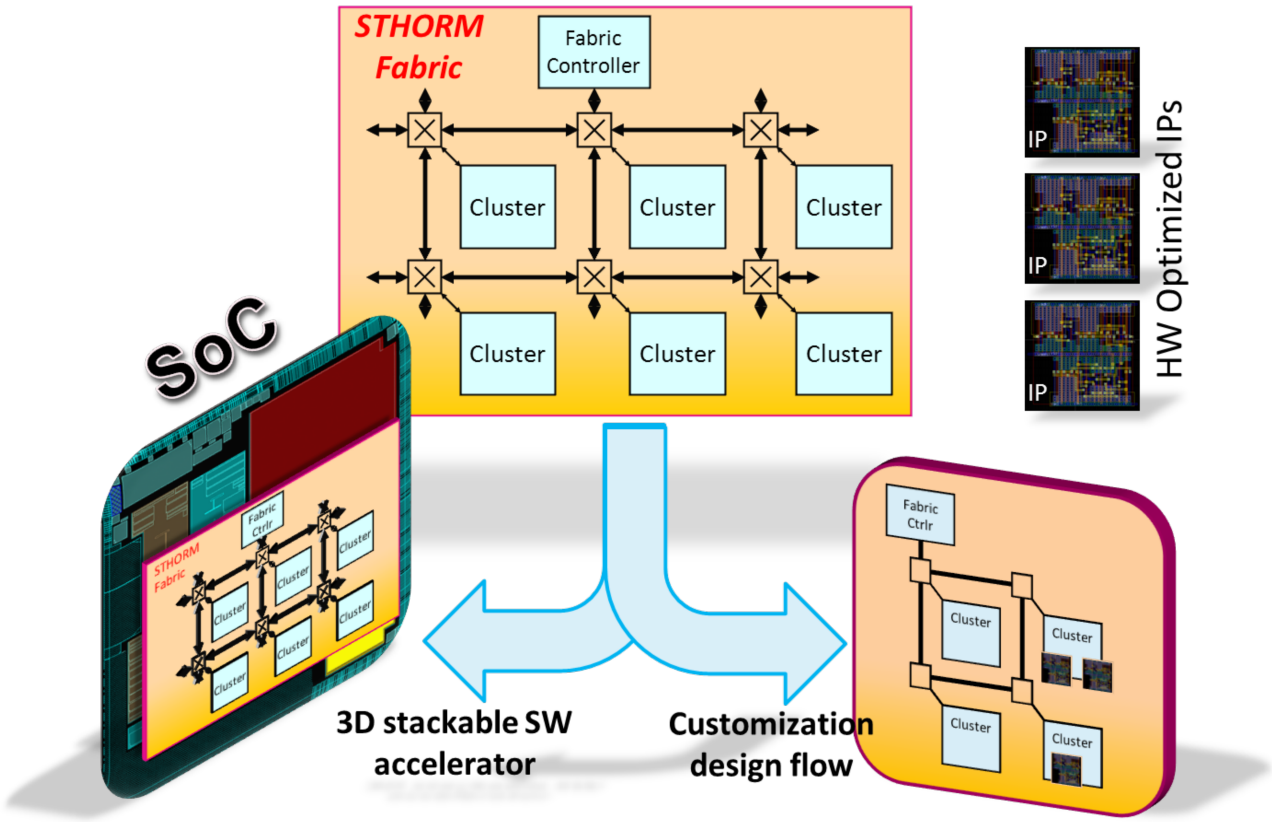


FIGURE 1.7 – *La fabrique de STHORM. Source: interne ST.*

La fabrique dispose de sa propre mémoire de deuxième niveau, ainsi que d'une mémoire externe qu'elle partage avec les autres modules du SoC. Le plan mémoriel est plat, ce qui signifie que l'ensemble des adresses, y compris celles des registres de périphériques, sont accessibles par tous les processeurs. Cependant, dans cette architecture mémorielle non uniforme, l'accès aux niveaux éloignés de la hiérarchie sont plus coûteux. Pour cette raison, les DMA sont là pour accélérer les transferts au sein de la fabrique et avec le reste du SoC.

Le contrôleur de la fabrique assure l'interface entre celle-ci et le SoC en communiquant avec son processeur hôte. Son rôle est de contrôler la procédure d'amorçage, ainsi que le placement et le chargement des applications.

La figure 1.8 illustre plus en détails la composition d'un groupe de calcul. Le sous-ensemble appelé ENCore<N> (où N est le nombre de SWPE) comprend des cœurs STxP70 en guise de processeurs programmables couplés à des mémoires caches d'instructions – mais pas de données – et des extensions pour les divisions et les évènements. Le reste englobe les HWPE et leur bus dédié, ainsi qu'un certain nombre de périphériques dont deux canaux DMA (*DMA channels*, DCHAN) et un module de synchronisation matérielle (*hardware synchronizer,* HWS).

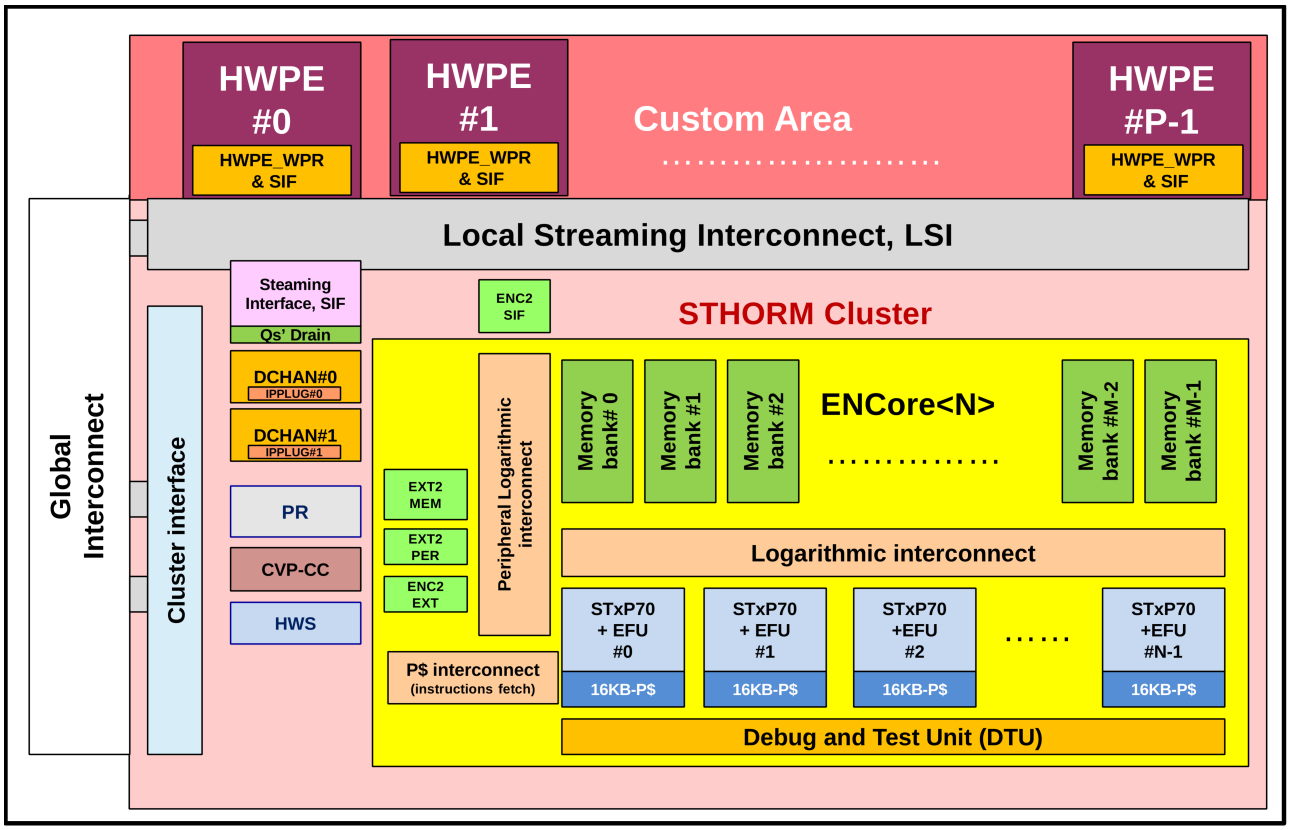


FIGURE 1.8 – *Un groupe de calcul de STHORM. Source: interne ST.*

1.3.2.1 Accélérateurs matériels et flux de données

STHORM a été conçu dans l'idée de pouvoir exécuter efficacement des applications à flux de données, c'est pourquoi les HWPE évoluent dans un espace distinct des SWPE où ce ne sont pas les instructions qui s'écoulent – comme dans le modèle de Von Neumann – mais les données. Pour cette raison, les HWPE disposent de leur propre bus d'interconnexion pour transporter les données sous forme de flux.

Dans le détail, un HWPE est constitué d'une enveloppe dont la structure est commune à toutes les instances, et d'une partie conçue par l'utilisateur qui dépend de la fonction à réaliser et contient les filtres à proprement parler. L'enveloppe comprend essentiellement les interfaces avec le bus, ainsi qu'un contrôleur. Ce dernier orchestre l'exécution des filtres *via* une machine à états et reçoit sa configuration de la part du logiciel par le truchement de registres mappés en mémoire. Celle-ci se fait en deux étapes :

1. sélection d'un jeu de filtres à exécuter ;
2. lancement simultané de tous les filtres sélectionnés.

Quand ceux-ci ont terminé leur exécution, le contrôleur écrit une valeur prédéfinie à une adresse mentionnée dans les registres *ad hoc*. Il est par exemple possible, par ce biais, de signaler au logiciel la fin d'exécution en demandant au synchroniseur matériel d'envoyer un évènement au SWPE initiateur.

1.3.3 Autres architectures similaires

Cette sous-section présente une sélection de quatre architectures présentant des caractéristiques comparables à STHORM. Pour une liste plus complète d'architectures hybrides de ce type, se reporter aux travaux de Yazdanpanah et coll. [17].

1.3.3.1 Mega-Leon

L'université de Bologne, qui a pris part à la conception de STHORM, a également développé pour ses propres recherches un groupe de calcul multicœur avec mémoire partagée et accélérateurs matériels étroitement couplés [18] très similaire. L'hétérogénéité repose ici sur l'association de processeurs généralistes SPARC-V8 – au lieu des STxP70 – et d'unités de calcul spécialisées directement connectées à la mémoire partagée. Le choix a ainsi été fait de privilégier un accès facilité à la mémoire de manière à pouvoir échanger efficacement entre SWPE et HWPE, au détriment d'une transmission optimisée entre accélérateurs. Pour assister la conception des unités de calcul matérielles, les auteurs proposent une méthode basée sur la synthèse de haut niveau.

1.3.3.2 FISC

Chen et coll. [19] proposent d'introduire dans le jeu d'instructions de processeurs généralistes des macro-opérations qui remplaceraient des fonctions par des appels à des accélérateurs (ensembles reconfigurables de modules câblés). La reconfiguration puis l'appel des HWPE ont lieu dans un étage dédié du pipeline des SWPE, après le décodage. Les HWPE sont étroitement couplés aux SWPE au sens où ils ont accès à la fois à la mémoire et aux registres. En matière de programmabilité, le matériel sous-jacent est caché au développeur par l'introduction d'une phase de compilation supplémentaire. On notera l'absence de support pour les flux de données.

1.3.3.3 Tartan

Mishra et coll. [20] présentent une architecture hybride qui comprend un SWPE et une fabrique hiérarchique reconfigurable sans horloge autorisant une exécution asynchrone dirigée par les données. L'application de l'utilisateur est scindée en deux parties : l'une s'exécutant sur le SWPE, l'autre sur la fabrique. L'espace mémoriel est partagé entre les deux unités de calcul. Le jeu d'instructions du SWPE doit être étendu avec des instructions de contrôle spécifiques. Le partitionnement de l'application, la génération du code pour le SWPE et la fabrique, ainsi que la synthèse de cette dernière sont effectués par le compilateur, de manière transparente pour l'utilisateur.

1.3.3.4 DySER

Cette dernière architecture [21] est constituée d'un tableau hétérogène de HWPE connectés par un réseau de commutateurs qui s'intègre dans le pipeline d'un SWPE. Comme STHORM, les communications se font par flux de données avec un protocole de transport à base de crédits. Ici encore, une extension du jeu d'instructions du SWPE est nécessaire pour contrôler et reconfigurer DySER. De même, le partitionnement de l'application et l'insertion des instructions spéciales sont faits par le compilateur.

1.4 Problème abordé

Suite aux observations formulées dans les sections précédentes concernant les caractéristiques des applications visées ainsi que celles des architectures ciblées, la problématique soulevée par cette

thèse peut s'exprimer de la manière suivante : comment exécuter efficacement une application à flux de données très dynamique sur une plateforme embarquée hybride ?

Cette question s'articule principalement autour de deux axes : le modèle d'exécution et l'ordonnancement. Dans le premier, il s'agit de faire correspondre un modèle adapté à une description à haut niveau d'abstraction des applications visées, à un modèle de programmation de bas niveau. Les modèles existants sont soit de haut niveau, ne prenant pas en compte les spécificités des architectures modernes – en particulier l'hybridité –, soit de bas niveau, ciblant une classe restreinte d'architectures voire une plateforme unique. Le modèle d'exécution présenté au chapitre 2 propose une solution intermédiaire située à mi-chemin entre ces deux pôles. Dans le second axe, au problème classique d'assignation et d'ordre en vue de minimiser la durée totale d'exécution, s'ajoute la contrainte liée à la capacité limitée de la mémoire. De plus, une grande part des applications pour l'embarqué étant caractérisées par une régularité permettant un ordonnancement statique peu coûteux, les solutions utilisées usuellement se prêtent moins à l'exécution efficace de programmes très dynamiques. Le chapitre 3 propose une solution fondée sur les heuristiques d'ordonnancement de liste et évalue les bénéfices potentiels d'un dynamisme accru ; le chapitre 5 propose une structure d'ordonnanceur compatible avec le modèle d'exécution du chapitre 2 et capable de supporter les formes les plus avancées de variabilité et d'imprédictibilité. En outre, deux contraintes supplémentaires doivent être introduites : la nécessité pour le logiciel système de rester aussi léger que possible, à la fois en matière de temps et d'espace, et la prise en compte de l'exécution en temps réel.

D'après ce qui précède, le problème abordé peut alors se formuler comme suit :

> ***Étant donné : une application à flux de données très dynamique décrite dans un modèle à haut niveau d'abstraction, une plateforme hybride embarquée et un ensemble de contraintes de temps ; trouver une exécution garantissant la sémantique de l'application, limitant l'empreinte mémorielle et respectant les contraintes temporelles.***

Chapitre 2. Modèles à flot de données

Il a été mentionné au chapitre 1 qu'un modèle de haut niveau était nécessaire pour décrire les applications à flux de données afin d'être en mesure de cibler des unités de calcul aussi bien matérielles que logiciels de façon efficace. Dans cette optique, le présent chapitre décrit les modèles de type flot de données (*dataflow*) et montre qu'ils représentent de bons candidats à cet égard. La première section expose les fondements théoriques puis énumère les solutions précédemment proposées, en particulier celle utilisée dans le cadre de STHORM ; la seconde présente la contribution de ce chapitre : un modèle d'exécution répondant à la problématique de la thèse ; enfin, la dernière conclut sur une réflexion concernant le risque d'interblocage.

2.1 État de l'art

Historiquement, toutes les recherches sur les modèles à flot de données font suite aux travaux fondateurs de Dennis [22] qui remontent à 1974. De nombreuses variantes – appelées modèles de calcul – ont depuis été proposées (voir les sous-sections suivantes pour un tour d'horizon de celles-ci) mais elles partagent toutes les mêmes principes de base suivants.

Les modèles à flot de données proposent de décrire chaque programme sous forme d'un graphe orienté dont les nœuds sont des entités de calcul fonctionnelles appelées *acteurs*, et les arcs sont des canaux de communication de taille conceptuellement infinie constituant la seule source de dépendance entre acteurs. Les données transitent d'un acteur à l'autre exclusivement par le biais des arcs, sous forme d'unités atomiques appelées *jetons* (*tokens*). La figure 2.1 donne un exemple simple d'un tel graphe où A, B, C et D sont quatre acteurs qui échangent des jetons de A vers D par l'intermédiaire de B et C.

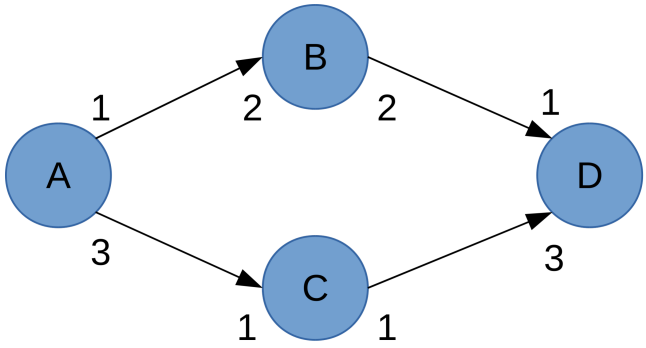


Figure 2.1 – *Exemple de graphe à flot de données comprenant quatre acteurs. Les chiffres aux extrémités des flèches indiquent les débits des ports.*

L'exécution d'un acteur, appelée *invocation* (*firing*), correspond à une séquence ordonnée et indivisible de trois étapes :

1. consommation d'un certain nombre de jetons sur les arcs entrants ;
2. application de la fonction de calcul sur les jetons consommés ;
3. production d'un certain nombre de jetons sur les arcs sortants correspondant aux résultats de la fonction de calcul.

Pour chaque acteur, un ensemble de règles d'activation (*firing rules*) définies par l'application déterminent sous quelles conditions celui-ci peut être invoqué : il s'agit, dans le cas général, du nombre et de la nature des jetons devant être présents sur les arcs d'entrée. Les interfaces des acteurs connectées à des arcs, et permettant donc le transfert de jetons, sont appelées *ports*. Le *débit* (*rate*) d'un port correspond au nombre de jetons transférés lors d'une invocation. Sur la figure 2.1, l'acteur A dispose de deux ports de sortie produisant respectivement 1 et 3 jetons, B dispose d'un port d'entrée et d'un port de sortie consommant tous deux 2 jetons, etc. Dans le cadre de cet exemple, si l'on suppose que les règles d'activation correspondent uniquement aux jetons consommés au niveau des ports d'entrée, alors l'acteur A est toujours activé, B nécessite 2 jetons, C en requiert 1, et D respectivement 1 et 3 sur chacun de ses ports.

Dans les modèles à flot de données, les lectures – c'est-à-dire l'opération consistant à tenter de consommer des jetons sur un arc donné – ne bloquent jamais grâce à l'existence des règles d'activation qui garantissent qu'une lecture n'aura lieu qu'à la condition que les jetons nécessaires soient présents (la section 2.1.2 présente un modèle analogue où les lectures sont bloquantes). De même, les canaux de communication étant supposés infinis, donc capables de stocker une quantité illimitée de jetons, les écritures ne peuvent pas non plus bloquer. On notera néanmoins que, en raison des contraintes d'implémentation, les écritures sont bloquantes en pratique : les canaux de communication doivent nécessairement avoir une capacité limitée dans un environnement où la mémoire est finie.

En outre, les nombreuses variantes des modèles à flot de données présentées dans les sous-sections suivantes disposent de certaines propriétés théoriques apportant des garanties spécifiques, parmi lesquelles :

- vivacité : absence d'interblocage intrinsèque au graphe (par opposition aux impasses artificielles introduites par le modèle d'exécution ou l'ordonnanceur) ;
- bornitude : possibilité pour un graphe donné d'être exécuté en mémoire finie ;
- cohérence : le nombre de jetons produits sur chaque arc tend vers le nombre de jetons consommés [23]
- terminaison : une exécution complète comprend un nombre fini d'opérations [24].

Les modèles de calcul dont les propriétés précédentes sont décidables sont généralement ordonnançables de manière *statique*, c'est-à-dire qu'il est possible de générer un ordonnancement correct à la compilation. Les modèles disposant d'une plus grande expressivité sont quant à eux ordonnançables de façon *quasi-statique* – c'est-à-dire qu'un ordonnancement peut être pré-calculé

à la compilation mais doit être ajusté à l'exécution – voire *dynamique* – tout le travail de l'ordonnanceur doit être mené à l'exécution. Dans l'exemple de la figure 2.1, il est possible de construire l'ordonnancement statique suivant: 2A 1B 6C 2D. Cette séquence peut être répétée indéfiniment avec une capacité de 2 jetons pour les files du haut et 6 jetons pour les files du bas, ce qui démontre à la fois la vivacité, la cohérence et la bornitude. La figure 2.2, en revanche, présente un exemple de graphe résultant nécessairement en une accumulation illimitée de jetons dans la file du bas: il n'est donc ni cohérent ni borné.

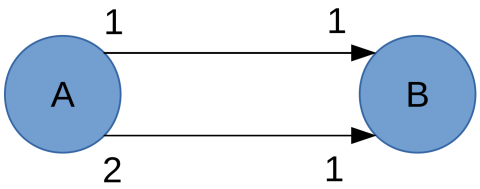


FIGURE 2.2 – *Exemple de graphe non borné.*

Les avantages de cette classe de modèles sont nombreux. Du point de vue de la description de l'application, la représentation graphique lui confère un aspect visuel intuitif. Le haut niveau d'abstraction rend la modélisation suffisamment générique pour pouvoir s'adapter à des cibles variées – il est par exemple possible de générer à partir d'un même graphe à flot de données du code dans différents langages aussi bien logiciels (p. ex. C ou Java) que matériels (p. ex. VHDL ou Verilog) ; à cela s'ajoute l'expression naturelle de la concurrence et l'exposition du parallélisme potentiel. Le découpage en acteurs facilite le partitionnement; de plus, il permet une programmation modulaire et incite à la réutilisabilité des composants, critère de choix pour les décodeurs vidéo (cf. chapitre 1). Du point de vue de l'implémentation, l'intérêt de décomposer le calcul en quantums indivisibles réside dans l'opportunité de s'affranchir de coûteux changements de contexte. Et le placement des acteurs sur des architectures parallèles telles que celles visées par cette thèse s'en trouve facilité.

Il convient néanmoins de souligner quelques inconvénients des modèles à flot de données qui empêchent leur adoption plus large. L'obstacle le plus manifeste est vraisemblablement le travail supplémentaire que constitue la modélisation de l'application. En effet, la majeure partie des programmes existants sont écrits dans un langage de bas niveau dont il est difficile d'extraire des représentations à plus haut niveau d'abstraction. C'est l'une des raisons d'être de la norme RVC mentionnée au chapitre 1. Par ailleurs, les modèles existants sont nombreux (cf. section 2.1.4) et souvent incompatibles les uns avec les autres. Le modèle d'exécution proposé à la section 2.2 permet de régler partiellement ce dernier problème.

En somme, il apparaît que les modèles à flot de données se prêtent bien à la description et à l'exécution des applications visées par cette thèse.

2.1.1 Paramètres et reconfigurations

On définit un paramètre comme un jeton particulier qui modifie les propriétés de l'acteur qui le reçoit. Comme le souligne Neuendorffer [25], les paramètres améliorent la réutilisabilité des acteurs en leur permettant d'être reconfigurés en fonction des valeurs qu'ils prennent, y compris en cours d'exécution. Il distingue les reconfigurations qui sont liées aux paramètres et celles,

structurelles, qui n'en dépendent pas. La section 2.2.2.2 présente une classification plus avancée des différents types de reconfigurations possibles dans le cas général. Dans tous les cas, ces reconfigurations ne peuvent avoir lieu qu'à des points de repos précis, entre deux invocations. L'auteur suggère trois possibilités de mécanismes de reconfiguration :

- automates finis : chaque transition correspond à une reconfiguration ;
- ports de reconfiguration : chaque paramètre est lié à un port par lequel les nouvelles valeurs sont reçues à chaque invocation ;
- acteurs de reconfiguration : un acteur spécifique est dédié à la reconfiguration de chaque paramètre, ce qui autorise une fréquence plus élevée qu'avec les ports.

De nombreux modèles de calcul à flot de données paramétrés ont été développés. Le plus ancien et le plus répandu est certainement PSDF [26] qui offre des débits paramétrables, garantit une exécution en mémoire bornée si les débits le sont, et permet un ordonnancement quasi-statique ; cependant, le modèle est très complexe, rendant la description d'une application malaisée, et ne permet pas des changements dynamiques de topologie. La figure 2.3 illustre cette complexité par un exemple de graphe dont l'objet est simplement la décimation des valeurs fournies en entrée par un facteur variant à chaque invocation en fonction d'un paramètre.

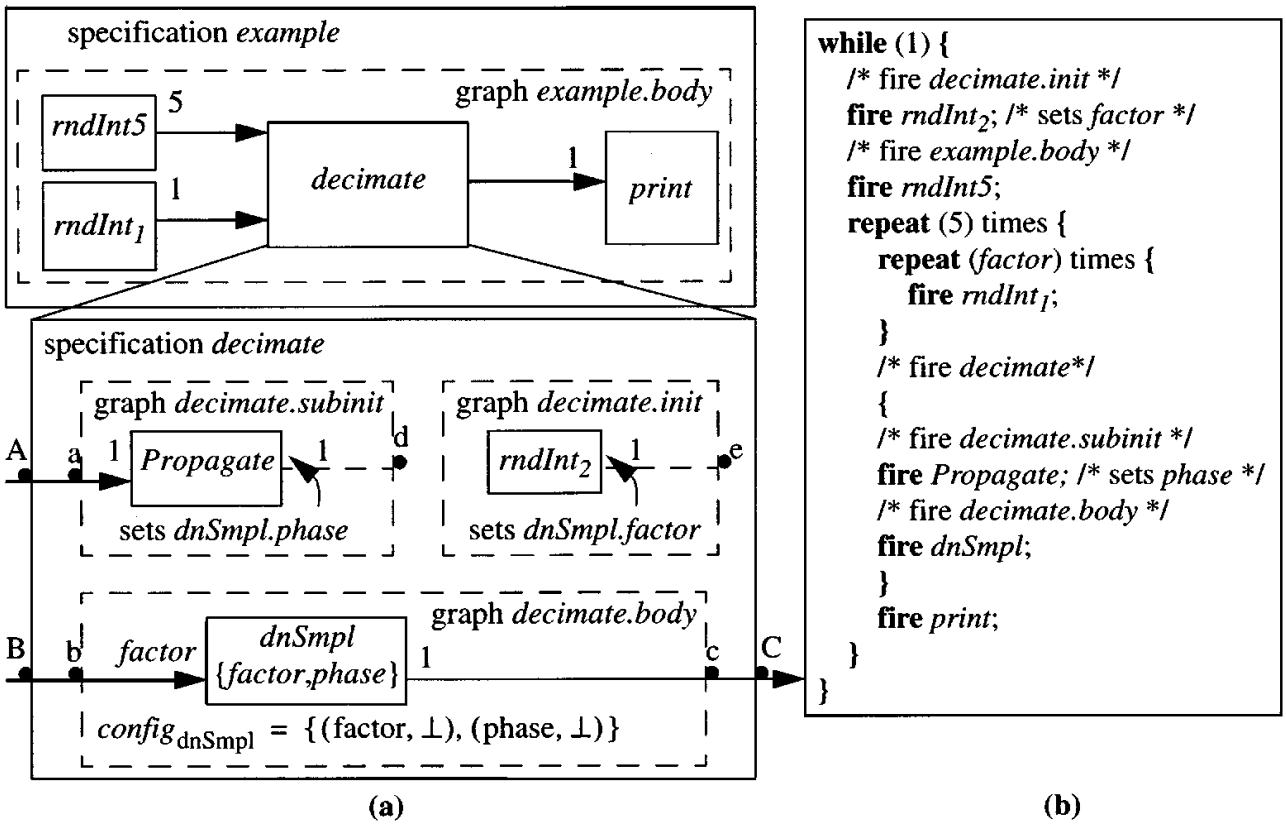


FIGURE 2.3 – *Exemple de spécification PSDF (a) et l'ordonnancement quasi-statique associé (b). Source : Bhattacharya et coll. [26].*

Plus récemment, SPDF [27] puis BPDF [28] ont tenté d'y remédier en proposant des modèles plus simples, permettant une analyse totalement statique et notamment une hiérarchisation automatique, contrairement à PSDF où celle-ci devait être réalisée manuellement par l'utilisateur. La figure 2.4 offre un exemple de graphe SPDF avec trois acteurs et deux paramètres (p et q). Bien

que BPDF ait hérité des paramètres booléens de BDF [29] – étendus aux entiers par IDF [30] – les garanties supplémentaires apportées viennent au détriment de l'expressivité : BPDF n'est pas Turing-complet, contrairement à BDF (cf. section 2.1.3).

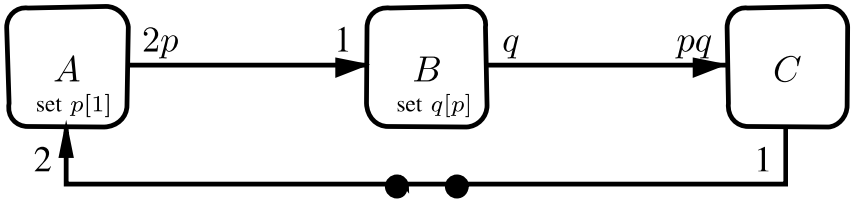


FIGURE 2.4 – *Exemple de graphe SPDF. Source : Fradet et coll. [27].*

2.1.2 Réseaux de processus

Il existe également une classe de modèles de calcul très similaires et qui se prêtent tout aussi bien aux applications à flux de données : les réseaux de processus (*process networks*). Cette section en présente les deux variantes les plus représentatives.

Les plus anciens travaux à ce sujet sont ceux de Gilles Kahn – qui a donné son nom au modèle KPN [31]. Dans un tel réseau, des processus concurrents communiquent *via* des canaux unidirectionnels. Comme le souligne Buck [24], ce peut être vu comme un ensemble de machines de Turing interconnectées par des bandes à sens unique, et où chaque machine dispose de sa propre bande. La comparaison est d'autant plus significative que les réseaux de processus de Kahn ont la même expressivité qu'une machine de Turing et, de ce fait, les mêmes questions indécidables, en particulier le problème de l'arrêt (il est impossible de déterminer en temps fini si une machine de Turing va s'arrêter). En effet, les questions de l'exécution en mémoire et en temps finis se ramènent toutes deux au problème de l'arrêt. De plus, dans le cas général, l'ordonnancement doit nécessairement être dynamique, et l'absence de règle d'activation peut rendre nécessaires des changements de contexte dans le cas où le nombre de ressources matérielles est inférieur au nombre de processus.

Malgré ces inconvénients, le modèle KPN dispose de propriétés théoriques intéressantes. Au niveau d'un processus, les lectures sont bloquantes – c'est-à-dire que tenter de lire un canal qui ne contient pas de données suspendra l'exécution jusqu'à ce que le processus à l'autre extrémité en produise –, ce qui permet de garantir le déterminisme du programme : quel que soit l'ordre dans lequel les processus sont exécutés, le résultat sera toujours le même.

Les modèles à flot de données étant très proches des réseaux de processus, un formalisme permettant de synthétiser les deux a été proposé. Les réseaux de processus à flot de données (DPN) peuvent être vus comme un cas particulier des KPN où les lectures sont rendues non bloquantes par l'introduction des règles d'activation. D'un point de vue formel, une séquence d'invocations d'un acteur est équivalente à un processus. Un DPN bénéficie donc de toutes les propriétés bénéfiques précédemment mentionnées, dont la possibilité de s'affranchir de coûteux changements de contexte. Un non-déterminisme limité est apporté, permettant par exemple de fusionner deux entrées de données dans un ordre quelconque. Le haut niveau d'expressivité que procure le modèle DPN, notamment en n'imposant aucune contrainte sur les débits, conduit aux

mêmes limitations que KPN en termes de décidabilité, l'une des conséquences pratiques étant, cette fois encore, le recours nécessaire à l'ordonnancement dynamique. Le tableau 2 synthétise les différences et similarités entre ces deux modèles.

	KPN	DPN
Écritures non bloquantes	oui	oui
Lectures bloquantes	oui	non
Règles d'activation	non	oui
Non déterminisme	interdit	autorisé

TABLEAU 2 – *Comparaison des propriétés théoriques des modèles KPN et DPN.*

En conclusion, il apparaît que le paradigme des réseaux de processus se prête bien à la description et à l'exécution des applications à flux de données, bien qu'ils fournissent peu d'outils d'analyse. Pour cette raison, le modèle d'exécution développé au cours de cette thèse et exposé à la section 2.2 se fonde à la fois sur DPN et KPN.

2.1.3 Modèles statiques

Bien qu'ils ne soient pas exploités directement dans les travaux présentés ici, un certain nombre de modèles analysables sont cités dans cette section, par souci d'exhaustivité et car ils occupent une place importante dans le spectre de la programmation à flot de données. Tous ces modèles ont en commun l'existence d'un concept d'itération intrinsèque – bien que les définitions diffèrent – qui autorise un ordonnancement statique – ou, dans le pire des cas, quasi-statique –, et des garanties fortes sur la cohérence et la vivacité, ou encore l'exécution en mémoire constante avec possibilité de dimensionner les tampons à la compilation.

SDF [32], l'un des premiers modèles statiques, est encore aujourd'hui l'un des plus utilisés, souvent avec de légères variantes. Sa propriété-clé est la constance des débits, qui est une hypothèse forte car elle restreint sensiblement l'expressivité mais offre en contrepartie toutes les garanties précitées au terme d'une analyse d'une relative simplicité. CSDF [33] reprend le même principe mais en autorisant une variation cyclique des débits; l'expressivité s'en trouve améliorée et la décidabilité – vivacité, cohérence et bornitude – demeure identique, tandis que l'analyse se complexifie. Ces deux modèles définissent une notion d'itération qui correspond à la plus petite séquence d'invocations permettant de retourner le graphe dans son état initial – c'est-à-dire que, pour chaque file, le nombre de jetons présents en début et en fin d'itération est le même. Cette propriété permet un ordonnancement statique cyclique. Enfin, BDF [29] procure davantage d'expressivité, avec notamment des débits variables qui autorisent une exécution conditionnelle dépendant des données et rendent le modèle Turing-complet, mais perd en décidabilité et en facilité d'ordonnancement.

2.1.4 Synthèse et comparaison des modèles existants

Comme l'ont montré les paragraphes précédents, les différents modèles de calcul à flot de données occupent un large spectre allant du plus statique – SDF – au plus dynamique – DPN –, comme le résume la figure 2.5.

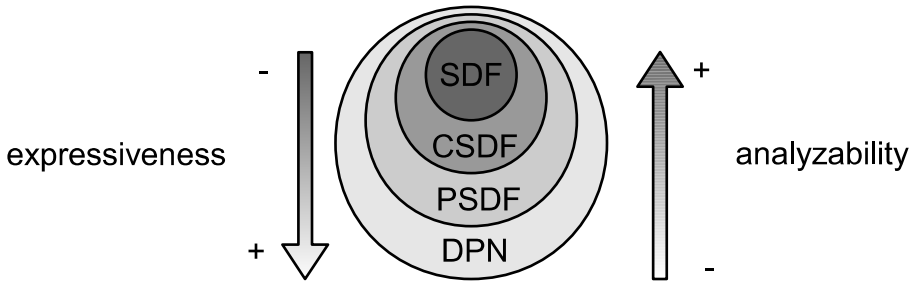


FIGURE 2.5 – *Comparaison de l'expressivité et de l'analysabilité des modèles de calcul à flot de données les plus répandus. Source: Wipliez [80].*

Les critères permettant de classifier ces modèles peuvent être étoffés jusqu'à englober quatre axes: expressivité, concision, décidabilité et efficacité d'implémentation [34]. Les deux premiers indiquent la restriction imposée sur la classe d'applications pouvant être représentée, et le niveau de compacité de cette représentation – ce second aspect prenant tout son sens lorsque la taille du programme augmente. La décidabilité (ou analysabilité) est déterminée par l'existence d'outils d'analyse permettant d'apporter des garanties sur l'exécution. Enfin, le dernier critère recouvre la complexité du processus d'ordonnancement et la taille du résultat. L'auteur arrive à la conclusion que, pour la plupart des modèles existants, la décidabilité et l'efficacité d'implémentation sont directement corrélées, de même pour l'expressivité et la concision; de plus, ces deux couples de critères varient en sens opposé.

Au terme de cet état de l'art, il apparaît qu'aucune solution existante n'est satisfaisante. Les modèles statiques manquent d'expressivité de manière patente: les contraintes sur les débits ne sont pas acceptables pour des applications très dynamiques. Les modèles paramétriques existants répondent partiellement à ce problème mais souffrent dans le même temps d'une complexité de mise en œuvre élevée qui cadre mal avec les objectifs de réactivité et de légèreté recherchés. Les réseaux de processus semblent plus adaptés mais DPN nécessite des règles d'activation pour chaque acteur – ce qui est incompatible avec certaines applications – et KPN nécessite un modèle d'exécution avec changements de contexte dès lors que plusieurs processus partagent la même ressource – ce qui induirait un surcoût important.

Néanmoins, selon ces observations, bien que ne répondant pas individuellement aux exigences, il est possible que, conjointement, les modèles DPN et KPN les satisfassent, comme le prouvera la section 2.2. Il est à noter que ce sont les plus expressifs et concis, en même temps que les moins analysables, mais que ce compromis est rendu nécessaire par la haute complexité et la forte variabilité des applications visées (cf. chapitre 1). En d'autres termes, DPN et KPN sont les seuls modèles à même de capturer tout le dynamisme des applications à flux de données telles que les décodeurs vidéo.

2.1.5 Modèles de programmation et outils de synthèse d'applications à flux de données

Les précédents paragraphes s'étant intéressés aux modèles à flot de données essentiellement d'un point de vue descriptif et représentatif, la présente section leur consacre une approche davantage tournée vers les questions de synthèse et d'implémentation. L'intégralité des développements et expérimentations menés au cours de cette thèse l'ayant été sur la plateforme STHORM présentée

au chapitre 1, l'accent est largement mis sur le modèle de programmation lui étant dédié. D'autres outils similaires sont présentés au chapitre 4.

2.1.5.1 PEDF: modèle de programmation pour STHORM

Ce modèle, initialement nommé de la sorte car l'exécution pouvait à l'origine être contrôlée à l'aide de prédicats (*predicated execution*), est librement inspiré des variantes dynamiques du paradigme flot de données présentées dans les sections précédentes. Il a été conçu dans l'idée d'exécuter des applications décrites sous forme de graphes hiérarchiques composés de *filtres*, matériels ou logiciels[12], pouvant être contrôlés par un ou plusieurs SWPE; ceux-ci ont la capacité de modifier le comportement de l'application durant son exécution, en démarrant ou arrêtant certains filtres et en modifiant leurs attributs de contrôle.

Les filtres sont les entités fonctionnelles primaires de PEDF: ce sont eux qui effectuent les calculs et le traitement des données. Chaque filtre (cf. figure 2.6) encapsule son propre état (S) et dispose de diverses interfaces: entrées (I) et sorties (S) de données, attributs de contrôle (A) pour modifier son comportement, et une gâchette (t) pour commander l'exécution. Les calculs sont effectués par une fonction de travail (W) appliquée aux données d'entrée, attributs et état pour produire les données de sortie. Au chargement et au déchargement de l'application, un constructeur (C) et un destructeur (D) sont appelés, respectivement.

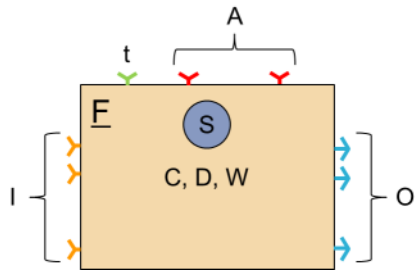


FIGURE 2.6 – *Filtre PEDF. Source: interne ST.*

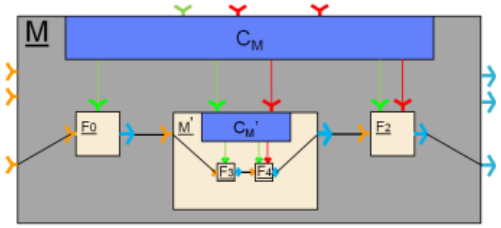


FIGURE 2.7 – *Exemple de composition hiérarchique dans PEDF. Source: interne ST.*

Un *module* est un ensemble comprenant au moins un filtre et exactement un contrôleur, et vu comme un filtre de l'extérieur, ce qui permet une composition hiérarchique (cf. figure 2.7). Chacun décrit les connexions de données entre les filtres qu'il contient ainsi que celles avec le contrôleur (en bleu sur la figure): gâchette, contrôles synchrone et asynchrone. Comme tout filtre, il dispose d'entrées, de sorties, d'attributs et d'une gâchette.

L'exécution est fondée sur une vague notion d'étape qui remplace le concept d'itération présent dans des modèles tels que SDF (cf. section 2.1.3). La quantité de données consommées par chaque filtre est variable et n'a pas besoin d'être connue pour qu'une nouvelle étape puisse être entamée; en d'autres termes, il n'y a pas de règles d'activation: PEDF n'est donc pas un modèle à flot de données à proprement parler. De plus, les lectures et les écritures sont bloquantes, ce qui rapproche davantage PEDF du modèle KPN. Pour garantir la cohérence de l'exécution, les attri-

12 Le support des filtres logiciels n'a, en réalité, jamais été implémenté par les concepteurs de PEDF. Le chapitre 5 consacre une section aux travaux menés à cet effet dans le cadre de cette thèse.

buts ne peuvent être lus et écrits par le contrôleur qu'entre deux étapes; le filtre y a quant à lui accès durant la totalité de son exécution.

Les contraintes sur les données imposent que leur taille et leur type soient connus à la compilation; ceci est également valable pour les attributs de contrôle et l'état des filtres. En revanche, une grande liberté est accordée quant à l'ordre dans lequel les données sont lues et écrites; il est en effet possible, dans une certaine mesure, d'accéder dans le désordre aux données au niveau des interfaces des filtres. Ce choix de conception, qui n'a pas d'utilité pratique pour les applications visées par cette thèse, a néanmoins de lourdes implications: non seulement il viole ouvertement le paradigme flot de données classique, qui impose explicitement que les transferts de données se fassent dans l'ordre, mais surtout cela amoindrit grandement l'efficacité des communications et alourdit inutilement le logiciel système pour le support d'un tel mécanisme qui, au surplus, n'est pas débrayable. Alors que, pour les flots de données classiques, une simple file avec lecture à une extrémité et écriture à l'autre suffit, la communication dans le désordre nécessite une structure de données plus complexe qui, dans le cas de PEDF, rend les accès coûteux, sujets à erreurs et difficiles à déboguer, du fait de la modélisation des flux en tableaux de jetons qui nécessitent d'incrémenter l'indice à chaque accès, d'une part, et du recours à une opaque combinaison de macros et de surcharge d'opérateurs, d'autre part. Par ailleurs, la diffusion – c'est-à-dire la connexion d'une interface de sortie à plusieurs interfaces d'entrée – est gérée, mais seulement entre deux filtres primitifs – pas à la frontière d'un module. Enfin, des prédicats permettent de désactiver certaines interfaces; dans ce cas, les accès sont non bloquants, les données écrites disparaissent et celles déjà présentes dans une connexion y demeurent jusqu'à la réactivation.

Le contrôle des filtres peut se faire de manière synchrone ou asynchrone. Dans le premier cas, une double banque de registres – analogue à un jeu de files de taille unitaire – permet de communiquer les valeurs des attributs pour l'étape N au cours de l'étape N-1, en vue de réduire les temps d'inactivité. Ce dispositif est asymétrique au sens où il ne fonctionne que dans le sens contrôleur-filtre; ce qui signifie que, dans l'autre sens, les valeurs des attributs écrites par le filtre écrasent systématiquement les précédentes. Il est par ailleurs possible de partager des attributs entre plusieurs filtres, bien que la gestion offerte par PEDF soit peu cohérente: dans le sens contrôleur-filtres, la propagation est garantie; en revanche, dans la direction opposée, celle-ci dépend de l'assignation des filtres aux ressources matérielles. Dans le cas asynchrone (ou dynamique), les interactions entre filtre et contrôleur sont autorisées au cours de l'exécution *via* un mécanisme équivalent au flot de données. Cette dernière fonctionnalité est problématique à deux égards: sur le plan théorique, elle constitue une violation du paradigme de reconfiguration (cf. section 2.1.1) par le simple fait de tolérer que celle-ci puisse avoir lieu en dehors des points de repos – en l'occurrence, pendant une étape –; sur le plan pratique, l'implémentation est boguée.

2.2 Modèle d'exécution proposé

Compte tenu des observations faites sur les modèles à flot de données existants, il apparaît clairement la nécessité d'une nouvelle approche mêlant les avantages des réseaux de processus – DPN et KPN – en termes d'expressivité, et ceux apportés par la reconfiguration paramétrique formalisée par Neuendorffer et mise en œuvre, entre autres, dans PSDF. Dans une démarche visant à exécuter efficacement des applications à flux de données caractérisées par une variabilité et une

imprédictibilité élevées – reconfigurations fréquentes, débits inconnus à l'invocation, temps d'exécution variant du simple au quadruple pour un même acteur, etc. – , sur une plateforme embarquée hybride, la conception d'un modèle d'exécution adapté s'impose. Le cahier des charges d'un tel modèle comprend les points suivants :

- être compatible à la fois avec les modèles descriptifs de haut niveau, y compris hiérarchiques et paramétrables, – de façon à pouvoir éventuellement bénéficier des garanties (en termes de correction, vivacité, etc.) qu'ils fournissent – et les modèles de programmation de plus bas niveau, en particulier PEDF ;
- capturer tout le dynamisme des applications visées, ce qui passe par un large support de tous les types de reconfigurations ;
- être simple à implémenter de façon efficace en milieu contraint ;
- garantir la sémantique de l'application.

Le modèle proposé reprend de nombreux concepts existants, inspirés de ceux mentionnés dans les sections précédentes, mais il introduit également un certain nombre de nouveautés détaillées à la section 2.2.2 : classification étendue des reconfigurations, composition DPN/KPN et paramètres indicatifs.

2.2.1 Description du modèle inspiré de l'existant

Le modèle d'exécution définit un ensemble d'actions et de règles visant à orchestrer l'exécution d'une application. Comme le montre la figure 2.8, il s'intercale dans la pile logicielle entre le compilateur et le logiciel système. De plus, il guide l'action de l'ordonnanceur. Le travail présenté dans la suite de cette section est mené sous l'hypothèse que les communications se font toujours dans l'ordre, aussi bien pour les données que pour les paramètres, conformément au paradigme flot de données classique et contrairement à ce que PEDF tolère, et ce par souci de simplicité et d'efficacité (cf. troisième point du cahier des charges).

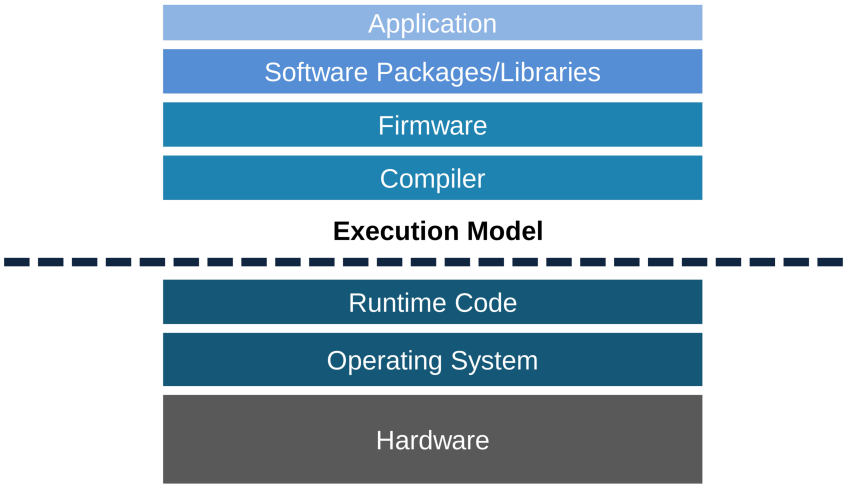


FIGURE 2.8 – *Place du modèle d'exécution dans la pile logicielle. Source : interne ST.*

Comme dans tout modèle à flot de données, l'unité de base est l'acteur, représenté par la figure 2.9. Dans ce cas, il est composite et formé de deux entités – un contrôleur et un filtre – dont

les attributions sont identiques à celles de PEDF. En première approximation, il s'agit donc d'une généralisation de PEDF où un acteur peut être vu comme un module ne comprenant qu'un seul filtre. Le modèle d'exécution n'étant pas lui-même hiérarchique pour des raisons de facilité d'implémentation et d'efficacité, les représentations dans des modèles de plus haut niveau doivent au préalable être aplaties ; le support intrinsèque de la hiérarchisation est néanmoins une perspective de travail intéressante car elle rendrait la description de l'application plus compacte. Les données sont transférées directement au niveau du filtre (*stream in* / *stream out*), tandis que les paramètres doivent être passés par l'entremise du contrôleur. L'invocation d'un acteur se fait en plusieurs étapes :

1. réception du signal `fire` par le contrôleur ;
2. collecte des paramètres par le contrôleur et reconfiguration du filtre ;
3. exécution (`run`) du filtre ;
4. envoi du signal `done` par le contrôleur pour indiquer que le filtre est prêt (`ready`) pour une nouvelle reconfiguration.

Le mécanisme de reconfiguration retenu (cf. section 2.1.1) est celui affectant à chaque acteur autant de ports *ad hoc* que de paramètres. Lors d'une invocation, chacun de ces ports consomme zéro ou un jeton de paramètre, conformément au mécanisme flot de données habituel et selon la règle d'activation en vigueur, avant que l'acteur ne procède à la reconfiguration adéquate.

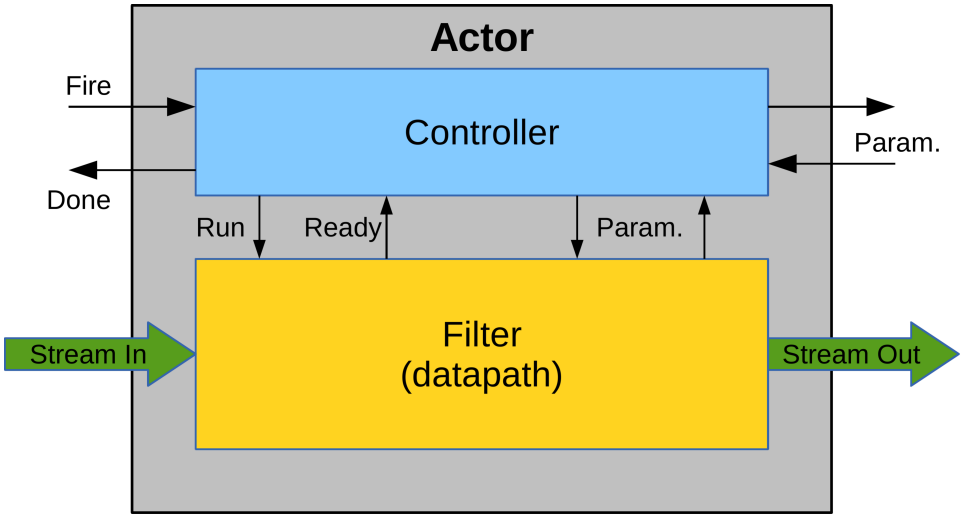


FIGURE 2.9 – *Architecture d'un acteur.*

Comme l'illustre la figure 2.10, la communication des paramètres s'effectue directement d'acteur à acteur, avec une file par consommateur mais sans file associée au producteur. Ainsi, sur cet exemple, l'acteur x produit un paramètre b qui est consommé par l'acteur y, lui-même générant un paramètre a à destination de son homologue. La diffusion d'un paramètre d'un producteur vers plusieurs consommateurs se fait par réplication dans chaque file de destination pour garantir l'indépendance des données et la fonctionnalité. La duplication – c'est-à-dire la production de la même valeur plusieurs fois en une seule invocation, utile par exemple quand le producteur et le consommateur ne sont pas invoqués à la même cadence – est optimisée pour limiter l'empreinte mémorielle : le paramètre produit est étiqueté avec le nombre d'exemplaires qui agit comme un

compteur décrémenté par le récepteur à chaque consommation, et il n'est retiré effectivement de la file que lorsque le compteur atteint zéro. Les règles d'activation exigent la présence du nombre de paramètres requis en entrée ; les valeurs sont indifférentes ; les données ne sont pas concernées. Le rôle de l'ordonnanceur vis-à-vis de ces règles est cantonné à leur vérification et leur application, le passage proprement dit des paramètres ne le concernant pas à ce stade. D'autre part, il a également la charge de l'émission des signaux `fire` pour ordonner l'invocation des acteurs et la réception des signaux `done` indiquant leur disponibilité.

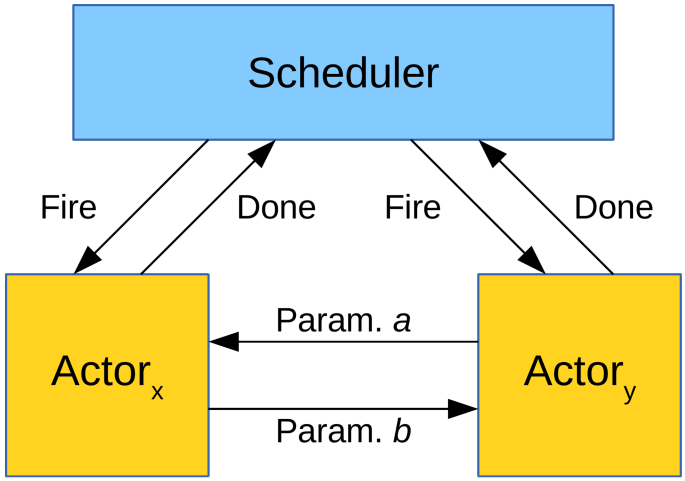


FIGURE 2.10 – *Vue d'ensemble du modèle d'exécution.*

2.2.2 Contributions

Cette section présente les nouveautés du modèle d'exécution par rapport à l'existant, ainsi que les contributions afférentes.

2.2.2.1 Sémantique de composition

Les modèles DPN et KPN disposent tous deux de propriétés intéressantes, mais dans des contextes différents. Dans le premier cas, il s'agit de contrôler que les données nécessaires à l'invocation sont disponibles avant de l'entamer ; dans le second, l'accent est mis sur l'exécution continue et le transfert des données dès leur disponibilité. Il est donc manifeste qu'en contrepartie le modèle DPN impose une connaissance supplémentaire de l'application pour permettre la spécification des règles d'activation, ce qui ne sied pas nécessairement à celles visées dans cette thèse. Ainsi, les implémentations des décodeurs vidéo ne fournissent pas toujours les informations nécessaires : c'est le cas notamment de celle utilisée à STMicroelectronics pour H.264, qui n'inclut dans sa version originale aucune indication sur les débits. C'est pour cette raison que le modèle KPN a été retenu pour régir l'exécution des filtres : il ne nécessite aucun renseignement supplémentaire quant à la nature et la quantité de données transférées. D'un autre côté, le nombre de paramètres échangés par les contrôleurs peut être connu, si ce n'est à la compilation, au moins en amont de chaque invocation durant l'exécution. Il est donc naturel de choisir le modèle DPN pour les contrôleurs afin de profiter des bénéfices supplémentaires qu'il apporte : décomposition en quantums de calcul permettant d'éviter les changements de contexte, et invocation uniquement dans le cas où les paramètres nécessaires sont disponibles de manière à réduire les possibilités de monopolisation des ressources par des acteurs inactifs. Sur la figure 2.9, la partie supérieure (contrôleur, échange des paramètres avec les autres acteurs, interaction avec l'ordonnanceur) appartient au

domaine DPN et la partie inférieure (filtre, entrées et sorties des flux de données) au domaine KPN.

Les règles de composition de ces deux modèles sont les suivantes. Au repos, c'est-à-dire hors invocation, le filtre n'a pas d'existence conceptuelle : l'acteur est donc uniquement composé de son contrôleur. Dans ces conditions, l'invocation de l'acteur est alors équivalente à celle du contrôleur. De ce fait, elle doit obéir à la sémantique DPN, donc à la règle d'activation de celui-ci ; en d'autres termes, l'invocation d'un acteur ne peut se faire que si le nombre de paramètres requis est présent en entrée. Elle se décompose ensuite en deux sous-invocations : la première permet au contrôleur de récolter les paramètres puis d'engendrer une instance de filtre correctement configurée, tandis que la seconde correspond à l'exécution du filtre. Durant toute la deuxième sous-invocation, le contrôleur disparaît conceptuellement et l'acteur se comporte alors comme un processus de Kahn. L'acteur revient dans son état DPN initial à l'issue de l'invocation.

La composition des modèles KPN et DPN dans le cas général a déjà été traitée, notamment dans le cadre du projet Ptolemy [35], mais celle proposée ici va plus loin.

2.2.2.2 Classification des paramètres

Comme l'a montré la section 2.1.1, le recours à la paramétrisation des modèles de calcul à flot de données est une pratique courante. Néanmoins, celle-ci n'est pas homogène au sens où la très grande variété des paramètres proposés recouvre des réalités très diverses. Pour y voir plus clair et comprendre en quoi ces paramètres sont utiles – voire indispensables – à la bonne exécution de l'application, il convient d'en établir une classification.

Neuendorffer [25] distingue uniquement les reconfigurations paramétriques et structurelles (cf. section 2.1.1). Dans cette optique, les paramètres ne sont en réalité que des attributs propres aux acteurs dont les valeurs changent au gré des reconfigurations. L'approche adoptée ici s'en démarque en définissant un paramètre comme un type de jeton, par opposition à ceux de données. Chacune des deux catégories dispose de ses propres ports et de sa propre sémantique, comme l'ont relaté les sections précédentes. Conceptuellement, les données subissent les traitements effectués par les acteurs, tandis que les paramètres les contrôlent par le biais des reconfigurations.

On peut alors classifier les paramètres selon les effets qu'ils produisent :

- fonctionnels : affectent le comportement interne du récepteur ;
- flot de données : changent le débit des ports de données du récepteur ;
- structurels : modifient la structure du graphe en activant ou désactivant des ensembles d'acteurs et de canaux ;
- méta-paramètres : changent le débit des ports de paramètres.

Il est à noter que la prise en compte des paramètres de flot de données n'est pas nécessaire en KPN pour garantir la correction de l'exécution. En revanche, ceux-ci peuvent fournir des informations utiles à l'ordonnanceur, comme le montre la prochaine section.

Les reconfigurations peuvent se faire à deux niveaux : acteur ou graphe. Les paramètres fonctionnels, flot de données et méta-paramètres concernent le premier cas ; il peut se produire à chaque

invocation. Les paramètres structurels, quant à eux, ne visent que le graphe; ces reconfigurations-là n'ont de sens qu'à des points de repos précis qui, dans le cas général, sont moins fréquents que les invocations des acteurs. Par exemple, pour le décodeur H.264, elles ne peuvent se faire que lorsque tous les acteurs sont au repos, entre deux trames.

2.2.2.3 Paramètres indicatifs

La classification décrite précédemment est insuffisante pour décrire pleinement le potentiel des paramètres, au sens où elle ne les considère que sous leur aspect essentiel à la bonne exécution d'une application. Or, il est souhaitable de porter un autre regard afin d'envisager également ceux dont l'intérêt réside aussi dans leur capacité à fournir des informations supplémentaires en vue d'améliorer l'efficacité de l'exécution. Les paramètres de flot de données appartiennent à cette seconde catégorie dès lors qu'ils obéissent au modèle KPN, celui-ci ne nécessitant pas de règles d'activation et donc le nombre de jetons consommés ne jouant pas de rôle particulier dans sa sémantique. Dans bien des situations, les paramètres fonctionnels peuvent eux aussi être mis à contribution de manière analogue.

On appelle *paramètre indicatif* tout paramètre dont la valeur peut être interprétée par le logiciel système en charge de l'application du modèle d'exécution – en particulier l'ordonnanceur – à des fins d'optimisation. Il est manifeste que cette définition recouvre une vaste classe d'attributs, dont la classification ci-dessus ne rend pas compte, pouvant renseigner sur des éléments qui ne seraient autrement pas représentables. Bien souvent, le concepteur de l'application dispose en effet d'une connaissance de celle-ci bien supérieure à ce que les modèles classiques lui permettent d'exprimer: quantité de calculs (paramètre fonctionnel), nombre de jetons transférés (paramètre flot de données), etc. Ces informations peuvent par exemple être utiles pour prévoir la durée d'une invocation, et ainsi prendre des décisions d'ordonnancement plus pertinentes. Dans ce cas, l'application doit fournir à l'ordonnanceur les éléments permettant d'interpréter les paramètres s'ils sont porteurs de telles informations. L'approche retenue dépend en grande partie de la nature et de la structure de l'application, mais on peut, à titre d'exemple, envisager un système dans lequel chaque valeur ou gamme de valeurs qu'un paramètre peut prendre doit être associé à un indicateur de durée dont la valeur est proportionnelle. Pour les stratégies d'ordonnancement qu'il est possible d'adopter dans de telles circonstances, se reporter aux chapitre 5 et 6.

2.2.3 Exemple d'application

Cette section expose et compare la représentation de l'application TNR (cf. chapitre 4 pour une présentation détaillée) dans un modèle de haut niveau et dans celui proposé.

La figure 2.11 illustre la modélisation SPDF de l'application. Les acteurs bleus sont ceux effectuant les calculs. Les paramètres, au nombre de sept, sont tous de type booléen (sauf n qui est entier) et contrôlent les débits de jetons mais aussi la structure du graphe en permettant la désactivation sélective de certains canaux. Le graphe comprend deux régions d'influence [27], l'une liée au paramètre a, l'autre aux paramètres b, e, n, p_1, p_2, et s. L'analyse permet de conclure à la vivacité et à la bornitude de l'application, et un ordonnancement quasi-statique peut être produit.

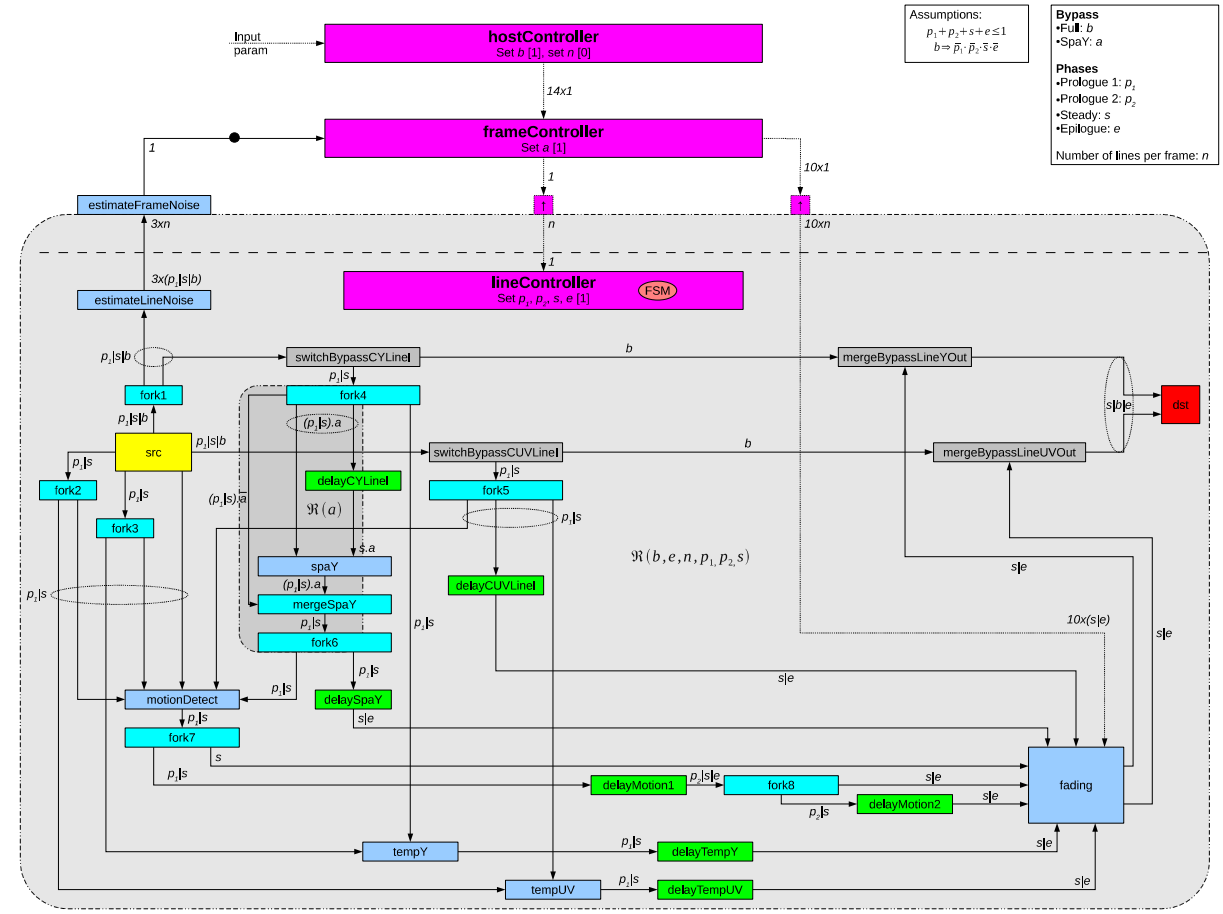


FIGURE 2.11 – *Modélisation SPDF de l'application TNR. Les paramètres sont listés en haut à droite. Les notations sont celles de Fradet et coll. [27]*

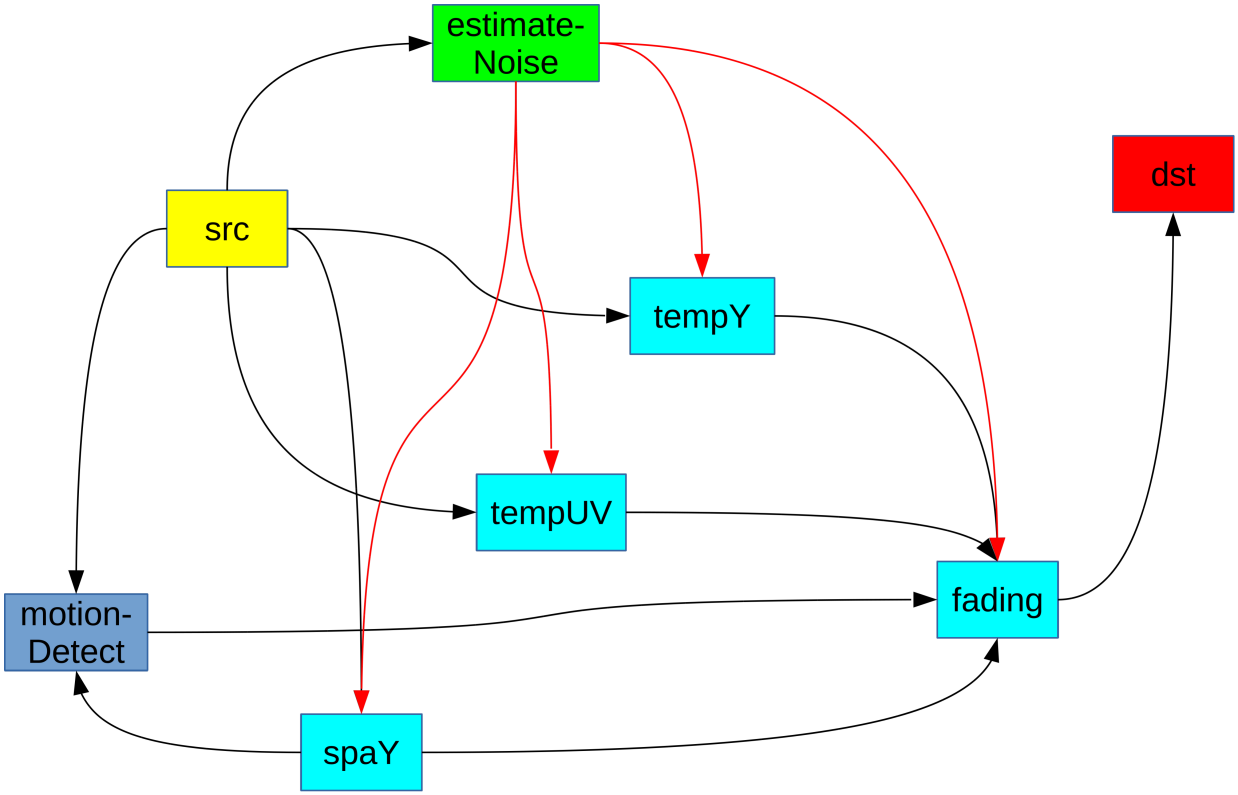


FIGURE 2.12 – *Représentation simplifiée de l'application TNR pour le modèle d'exécution proposé.*

La figure 2.12 représente la même application de façon conforme au modèle d'exécution proposé; seuls les acteurs principaux apparaissent. Les flèches rouges correspondent aux canaux de communication dédiés au seul paramètre fonctionnel. On notera en particulier l'absence d'indication quant aux débits des ports. Par conséquent, aucune analyse statique n'est possible.

2.3 Réflexion sur la gestion du risque d'interblocage

Dans les réseaux de processus, il existe deux types d'impasses dues à des blocages: applicative, dans le cas d'une mauvaise conception de l'application, ou artificielle, introduite par le modèle d'exécution [36]. Dans le domaine DPN, les règles d'activation garantissent l'absence de blocage artificiel sur des jetons entrants, et, par une simple extension du modèle, la connaissance des débits de production – notamment par le biais des paramètres indicatifs décrits à la section précédente – permet également d'éviter d'éventuels blocages sur des files de sortie pleines. Dans le domaine KPN, en revanche, hormis si les débits sont fournis sous forme de paramètres indicatifs, il n'existe pas de telles garanties. Un exemple typique de blocage artificiel est le sous-dimensionnement d'une ou plusieurs files inter-processus. Un certain nombre d'algorithmes [37]–[39] ont été proposés, de complexités diverses, pour tenter de venir à bout de ces problèmes. L'objet de cette section n'est pas d'en ajouter un nouveau à la liste, mais de présenter une discussion sur les méthodes de détection et de résolution pratiques qui pourraient être mises en œuvre sur des plateformes telles que celles visées par cette thèse.

À titre de rappel, la question de la détection ne se pose réellement que pour les implémentations à plusieurs unités de calcul. En effet, dans le cas simple monoprocesseur, la solution classique consiste à ordonnancer chaque processus dans son propre fil d'exécution *(thread)* aux côtés d'un fil supplémentaire doté d'une priorité inférieure, de telle sorte que celui-ci n'obtienne la main qu'en cas de blocage de tous les processus, signe que l'exécution de l'application a abouti à une impasse, ce fil peut alors lancer la procédure de résolution. Dans le cas multiprocesseur, en revanche, des synchronisations supplémentaires entre PE s'imposent, *a minima*. On peut distinguer deux approches: distribuée ou centralisée. Dans la première, chaque PE est capable de détecter et signaler ses propres blocages et déblocages; lorsque tous les PE ont signalé leur blocage pendant suffisamment longtemps – pour assurer la dissipation des effets transitoires dus, entre autres, à la latence du réseau –, la procédure de résolution est lancée. Ce système présente des inconvénients: l'émission des signaux consomme de potentiellement nombreux cycles de processeur et risque d'encombrer fortement le bus. Dans la deuxième approche, un PE dédié surveille en permanence l'état de toutes les files inter-processus à la recherche de chaines causales problématiques. Cette solution peut être envisageable à condition de mutualiser le PE en charge de la surveillance; dans STHORM, par exemple, le contrôleur de la fabrique, qui est essentiellement inactif en dehors des phases de chargement et de déchargement de l'application, pourrait assurer cette tâche si les files matérielles exposaient leur état au logiciel.

Dans tous les cas, les algorithmes ou les mécanismes sont coûteux à mettre en œuvre à l'exécution, ou nécessitent un support architectural lourd. Une solution de contournement praticable serait le recours à un chien de garde initialisé par l'application qui interrompt la commande en cours et remet la plateforme dans un état cohérent à chaque fois que les contraintes de traitement en temps réel sont violées.

2.4 Conclusion

Ce chapitre a présenté un modèle d'exécution fondé sur la théorie des réseaux de processus et des flots de données, à même d'apporter une réponse à la problématique soulevée par cette thèse en matière d'exécution d'applications à flux de données sur des architectures hybrides. Une des nouveautés de ce modèle est l'introduction de paramètres indicatifs (facultatifs) visant à améliorer l'efficacité de l'ordonnancement. Cette contribution a été mise en perspective vis-à-vis des travaux existants – très fournis dans le domaine de l'analyse statique au prix d'une expressivité insuffisante pour les applications visées –, notamment *via* l'exemple d'une application réelle. D'autre part, une classification plus complète que l'existante des paramètres dans les modèles à flot de données a été présentée. La question du risque d'interblocage consubstantielle aux réseaux de processus a également été évoquée et des solutions concrètes ont été proposées. Les concepts abordés ici seront mis en pratique aux chapitres 5 et 6. Le chapitre 3 s'intéresse quant à lui à l'ordonnancement sous contrainte mémorielle des applications à flux de données.

Le chapitre 5 montrera que l'absence de règles d'activation pour les filtres peut rendre l'ordonnancement impossible. Ce choix de conception peut, à certains égards, paraître surprenant, mais il découle d'une approche pragmatique considérant que l'effort d'analyse nécessaire à la mise au point de ces règles est trop important en regard des bénéfices qu'il apporte et surtout des contraintes extérieures liées au contexte industriel (temps de mise sur le marché, etc.). Ainsi, vu la complexité de l'implémentation du décodeur H.264 mise à disposition par STMicroelectronics, s'il avait fallu l'exécuter intégralement en DPN, des modifications en profondeur de la structure de l'application auraient été nécessaires, ce qui aurait constitué une tâche supplémentaire chronophage.

Chapitre 3. Ordonnancement de liste sous contrainte de mémoire

Comme il a été vu précédemment, seul un environnement de calcul parallèle serait à même de répondre aux besoins de puissance croissants des codecs vidéo récents. Or le recours au parallélisme soulève un certain nombre de problèmes, absents en cas de traitement séquentiel, dont celui de l'ordonnancement : sur quel processeur et dans quel ordre exécuter chaque tâche ? Pour y répondre, de nombreux algorithmes existent déjà, mais ils sont en grande majorité destinés au calcul haute performance sur des (ensembles de) stations où la mémoire n'est généralement pas une contrainte, contrairement au monde de l'embarqué dans lequel réduire l'empreinte d'un programme est une préoccupation majeure. Néanmoins, les solutions traditionnellement issues de l'informatique haut niveau, telles que les heuristiques d'ordonnancement dites « de liste », démontrent pour certaines de bonnes performances alliées à une faible complexité, qui se prêtent donc bien à des systèmes légers à l'instar de ceux visés par cette thèse.

C'est pourquoi l'objet de ce chapitre est l'extension d'une large classe d'algorithmes d'ordonnancement de liste par l'introduction de contraintes mémorielles au processus de décision. Faisant suite au constat selon lequel l'application brutale des contraintes mémorielles produit de piètres résultats, voire dans certains cas des interblocages, la principale contribution présentée ici est l'élaboration d'un procédé qui, d'une part, garantit la vivacité de l'exécution et, d'autre part, facilite la recherche d'un compromis entre durée totale d'exécution et empreinte mémorielle. Une méthode visant à guider le positionnement de la stratégie d'ordonnancement sur le spectre allant du statisme au dynamisme est également décrite.

Dans le détail, le reste du chapitre s'articule comme suit : la section 3.1 discute des travaux connexes ; la section 3.2 décrit les modèles utilisés ; la section 3.3 définit formellement le problème abordé ; la section 3.4 présente le cœur de la contribution, à savoir une méthode visant à adapter les priorités affectées par les heuristiques d'ordonnancement de liste en prenant en considération les contraintes mémorielles ; la section 3.5 relate les expérimentations menées et discute des résultats obtenus.

3.1 État de l'art

L'ordonnancement d'applications sur des machines parallèles est un problème NP-difficile y compris dans le cas où ces dernières sont homogènes [40], ce qui justifie le recours à des heuristiques. On répartit traditionnellement celles-ci en deux catégories selon qu'elles sont statiques – elles s'exécutent avant le chargement du programme – ou dynamiques – elles opèrent en même temps que celui-ci. Lee & Ha [41] notent qu'en réalité le travail d'un ordonnanceur peut se décomposer en trois opérations distinctes :

1. l'assignation des tâches aux processeurs ;
2. la spécification de l'ordre d'exécution sur chaque processeur ;
3. la détermination de la date de départ de chaque tâche.

Il en résulte que la classification peut s'étoffer jusqu'à comprendre quatre catégories (cf. tableau 3) :

- totalement dynamique : les trois opérations ont lieu à l'exécution ;
- assignation statique : seule l'allocation est faite à la compilation ;
- auto-séquencé : l'assignation et l'ordre sont prédéterminés, mais le programme détermine lui-même à l'exécution la date où il peut lancer la tâche suivante ;
- totalement statique : les trois activités se font à la compilation.

	Assignation	**Ordre**	**Date de début**
Totalement dynamique	Exécution	Exécution	Exécution
Assignation statique	Compilation	Exécution	Exécution
Auto-séquencé	Compilation	Compilation	Exécution
Totalement statique	Compilation	Compilation	Compilation

TABLEAU 3 – *Répartition des activités d'un ordonnanceur (en haut) en fonction de la stratégie (à gauche).*

Le choix de la stratégie s'effectue selon le compromis suivant : plus un ordonnanceur sera statique, moins il induira de coûts dans l'implémentation, mais moins il sera efficace dans le cas de tâches à durées non déterministes ; au contraire, plus un ordonnanceur sera dynamique, plus il sera à même de prendre en compte d'éventuelles variations dans la durée des tâches, mais plus il sera coûteux à mettre en œuvre. Dans le cas de codecs vidéo tels que H.264 et HEVC, un ordonnanceur totalement statique est d'emblée exclu car la durée des tâches est soumise à de fortes variations ; à l'opposé, un ordonnanceur totalement dynamique aurait un coût relativement lourd pour un système embarqué et devrait donc être circonscrit aux cas où il a été démontré que toutes les autres stratégies échouent à fournir la qualité de service requise. L'ordonnancement totalement dynamique n'est pas considéré dans ce chapitre. Une discussion plus approfondie sur le choix du positionnement dans le spectre statique-dynamique sera présentée à la section 3.5.1.

En outre, un paramètre supplémentaire est à prendre en compte dans la mise au point d'un ordonnanceur : le type d'algorithme. Celui-ci peut appartenir notamment à l'une des catégories suivantes [42] :

- les heuristiques d'ordonnancement de liste, décrites ci-après ;
- les heuristiques de regroupement, dont le principe est de former puis de fusionner des groupes de tâches de telle sorte que toutes celles appartenant à un même groupe soient exécutées sur le même processeur ;
- les heuristiques de duplication de tâches qui consistent à assigner certaines tâches de façon redondante en vue de réduire les communications inter-processeurs.

Les heuristiques d'ordonnancement de liste à priorités statiques étant généralement peu complexes et relativement performantes [43], ce sont elles qui ont été retenues pour cette étude. On notera en particulier que, lorsque les durées des tâches sont connues – ou, à défaut, qu'elles peuvent être estimées – un ordonnancement de liste, même simpliste, est presque optimal en termes de durée totale d'exécution[13] [43] et au plus deux fois plus long qu'un ordonnancement optimal [44]. Le principe est d'assigner des priorités aux tâches à exécuter et de les placer dans une liste ordonnée par priorité décroissante ; parmi les tâches disponibles, la première à être traitée sera toujours celle ayant la plus haute priorité, c'est-à-dire la première de la liste. En cas d'égalité, les *ex æquo* sont départagés aléatoirement. L'assignation à un processeur se fait de façon à minimiser une fonction de coût prédéfinie.

3.1.1 Heuristiques existantes

Dans le cas hétérogène, de nombreux algorithmes ont été proposés dans la littérature (se reporter à Canon et coll. [45] pour l'étude d'une vingtaine d'entre eux). Parmi ceux-ci, HEFT (*Heterogeneous Earliest Finish Time*) [42] fait preuve de l'un des meilleurs rapports performance-complexité. Cette heuristique détermine un ordonnancement totalement statique d'un graphe de tâches sur un environnement hétérogène de manière à minimiser la durée totale d'exécution de l'application. Il est possible de le convertir en ordonnancement auto-séquencé, voire à assignation statique, en éliminant les informations que l'on souhaite déterminer à l'exécution [41]. L'algorithme est constitué de deux phases principales : détermination des priorités, puis sélection des processeurs. La première étape consiste donc à affecter lesdites priorités aux tâches à ordonnancer. Pour ce faire, l'application est décrite sous forme d'un graphe acyclique orienté dont les nœuds représentent les tâches[14] et les arcs les dépendances entre elles. Les premiers sont pondérés par les durées d'exécution moyennes, et les seconds par les coûts de communication moyens. La priorité d'une tâche correspond à son rang ascendant dans le graphe, c'est-à-dire la somme des poids sur le chemin critique du nœud au puits du graphe, y compris ceux-ci. La seconde étape peut alors s'effectuer comme suit : lorsqu'une tâche est retirée de la liste, l'algorithme recherche le processeur capable de minimiser sa date de fin – y compris en tirant parti des temps d'inactivité laissés à ce stade par l'ordonnanceur – et la lui alloue dans la mesure où toutes les dépendances de données sont satisfaites.

HEFT est particulièrement adapté dans le cadre de cette étude sous plusieurs aspects : sur le plan embarqué, il s'agit d'un algorithme à faible complexité, facile à implémenter et qui, pour cette raison, se prête bien à une stratégie partiellement dynamique : sur le plan hétérogène, il prend en compte les variations de vitesse d'exécution entre les ressources de calcul en fonction des tâches ; sur le plan applicatif, les algorithmes de décodage vidéo, notamment, sont facilement représentables sous forme de graphes de tâches. Néanmoins, il présente un certain nombre de limites : l'hypothèse est faite qu'un processeur quelconque peut exécuter une tâche quelconque, ce qui n'est pas le cas dans le modèle retenu (cf. section 3.2.1) ; lorsque la quantité de données échangées entre deux tâches est importante et que celles-ci sont assignées à des ressources de calcul éloignées, le coût peut être significativement plus élevé que si les deux étaient allouées sur le même processeur en dépit d'une vitesse d'exécution inférieure ; et, surtout, la notion de mémoire en est absente.

13 C'est-à-dire à 5 % de l'optimal dans au moins 90 % des cas.

14 De ce fait, les termes *tâche* et *nœud* seront employés de façon interchangeable dans la suite de ce chapitre.

SDC (*Scheduling for processors with Different Capabilities*) [46] résout ces deux premiers points en prenant en compte la rareté des ressources – le poids d'un nœud considère non seulement le coût d'exécution moyen mais aussi le pourcentage de processeurs compatibles – et en incorporant les descendants directs dans le calcul de la fonction objective. PEFT (*Predict Earliest Finish Time*) [47] fait un constat analogue et va même encore plus loin en considérant le coût d'exécution de tous les descendants jusqu'au nœud de sortie. Par ailleurs, il existe des heuristiques d'ordonnancement de liste à priorités dynamiques : c'est le cas de DLS (*Dynamic Level Scheduling*) [48] où les priorités varient au cours du processus d'ordonnancement. Cette classe d'heuristiques a été exclue de la présente étude car, bien que pouvant se prévaloir de bons résultats, elles souffrent de temps d'exécution conséquents [42].

En ce qui concerne les contraintes de mémoire, les premiers travaux remontent aux problèmes d'allocation de registres [49] dont les solutions récentes sont quasiment toutes fondées sur les techniques de coloration de graphe [50], [51]. Il existe également des procédés visant à optimiser l'empreinte de l'ordonnancement par lots [52]. De plus, il est établi qu'optimiser la durée totale d'exécution sous contrainte de ressources est NP-difficile pour presque tous les problèmes non triviaux [40]. Des solutions spécifiques à certaines applications, comme les solveurs directs de matrices creuses, ont été proposées [53]. Des travaux plus récents [54] ont étudié la parallélisation à partir de graphes de tâches en forme d'arbres ciblant l'utilisation de la mémoire et la durée totale d'exécution ; une extension [55] du modèle permet de traiter des structures arbitraires. Enfin, en matière d'ordonnancement temps-réel, Baker [56] présente une méthode prenant en compte une capacité mémorielle limitée, mais elle s'applique à des tâches préemptibles avec dates limites d'exécution, ce qui n'est pas compatible avec les hypothèses retenues dans le cadre de cette thèse.

En conclusion, il apparaît que les études et solutions manquent pour s'attaquer à l'ordonnancement d'applications sur les systèmes embarqués par le biais de techniques efficaces, telles que l'ordonnancement de liste, tout en portant attention aux contraintes mémorielles et à la variabilité des durées d'exécution des tâches. L'objet de la suite de ce chapitre est de combler ce manque.

3.2 Définitions et modèles

Cette section expose de façon plus détaillée le contexte et les hypothèses de travail propres à ce chapitre : le modèle de la plateforme cible, le déroulement de l'exécution et les contraintes de mémoire.

3.2.1 Environnement de calcul

En vue d'être en mesure de tirer parti des heuristiques classiques d'ordonnancement telles que HEFT, tout en restant suffisamment général pour être appliqué à la majorité des architectures embarquées, le modèle de plateforme décrit au chapitre 1 doit s'adjoindre un certain nombre d'hypothèses simplificatrices :

– La plateforme est composée de plusieurs PE indépendants. Pour une tâche donnée, tous les PE ne sont pas aussi efficaces, certains peuvent même être inaptes à l'exécuter. Par exemple, chaque HWPE peut uniquement accomplir la tâche pour laquelle il a été conçu,

et les tâches de transfert mémoriel ne sont en mesure d'être traitées que par le contrôleur DMA, dont c'est la seule attribution.

- Les données sont initialement placées en mémoire externe et doivent nécessairement être transférées en mémoire locale par l'entremise du DMA, de façon à ce que l'application puisse travailler dessus.
- Pour exécuter les tâches, les PE accèdent aux données situées en mémoire locale. La latence et le coût de ces accès sont supposés sans contention et compris dans la durée des tâches.

La première hypothèse n'est en vérité pas une simplification : elle indique seulement le fonctionnement de la plateforme STHORM. La seconde reflète la façon dont les applications ciblées (comme les algorithmes de traitement d'images) sont typiquement implémentées sur de telles architectures pour des raisons de performance. La dernière est la seule véritable simplification : les accès à la mémoire locale ne peuvent généralement pas être garantis sans contention sur des plateformes réelles. Néanmoins, le surcoût induit par celle-ci peut être négligé dans la plupart des cas.

De plus, pour permettre l'introduction de considérations sur la mémoire, il convient de raffiner le modèle de celle-ci. La mémoire locale, étroitement couplée aux PE, rapide mais de taille restreinte, a une capacité exprimée en nombre d'*emplacements*. La mémoire externe est quant à elle considérée de taille infinie.

3.2.2 Modèle d'exécution

Comme il a été vu au chapitre 2, les applications pour STHORM obéissent habituellement à un modèle de calcul de type flot de données. Dans le présent chapitre, le modèle d'exécution repose non plus sur une représentation flot de données mais sur un graphe de tâches. Pour pouvoir ordonnancer des applications décrites sous forme de graphes à flot de données, il est donc nécessaire de procéder à une transformation. Celle-ci consiste simplement à dérouler plusieurs itérations du graphe à flot de données en simulant et construisant les tâches respectives et leurs dépendances. Le nombre d'itérations à instancier dépend des facteurs suivants. D'un côté, plus il y aura d'itérations, plus grand sera le graphe de tâches et meilleure sera la compréhension de l'application, auquel cas il sera d'autant plus facile de prendre des décisions d'ordonnancement pertinentes. D'un autre côté, le graphe de tâches peut croître à tel point que le temps d'ordonnancement risque de devenir insoutenable. Pis, la taille du résultat pourrait excéder la capacité de la mémoire disponible pour le stocker sur la cible embarquée. La solution consiste à trouver un compromis entre la qualité de l'ordonnancement produit et sa taille. Une telle décision est laissée à la charge du concepteur de l'application. Techniquement, il est néanmoins possible d'appliquer le même ordonnancement fenêtre par fenêtre comme si le graphe à flot de données était déroulé dynamiquement.

3.2.3 Modèle de mémoire

Afin de prendre en considération l'aspect mémoriel, il convient d'introduire un nouveau type de tâches dédié : les allocations et les libérations d'emplacements de mémoire tels que définis à la section 3.2.1. Une fois qu'un emplacement a été alloué par une tâche d'allocation, sa référence est

passée d'acteur en acteur sous forme de jeton, jusqu'à la tâche libératrice. Ce type de tâches ne peut être traité que par un SWPE, et leur ordonnancement est plus complexe que la normale. En effet, lorsqu'elle est exécutée, chacune d'entre elles peut soit consommer soit libérer une quantité donnée de mémoire locale exprimée en nombre de jetons. De manière à ce que le modèle demeure simple, on suppose – sans perte de généralité – qu'un emplacement peut accueillir exactement un jeton. Le nombre de jetons transférés, et donc le nombre d'emplacements consommés ou libérés, s'exprime comme un coût algébrique: positif s'il s'agit d'une allocation, négatif s'il s'agit d'une libération. Le nombre d'emplacements disponibles est mis à jour à chaque tâche exécutée en y soustrayant algébriquement son coût; il ne peut jamais être négatif: lorsqu'il s'annule, il faut d'abord ordonnancer une tâche libératrice avant d'en exécuter une autre consommatrice.

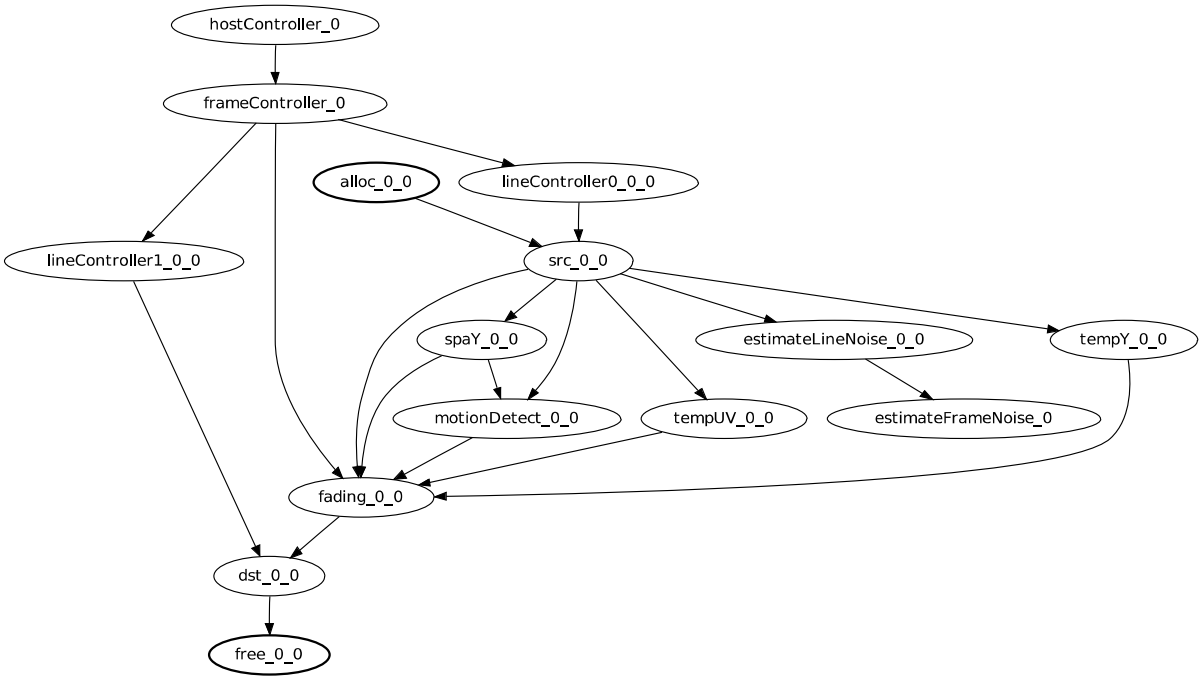


FIGURE 3.1 – *Exemple de graphe de tâches pour l'application TNR. Une seule ligne est traitée. Pour n lignes, il faut exécuter les tâches à double suffixe n fois. Les tâches gérant la mémoire apparaissent en traits plus épais:* `alloc_0_0` *consomme de la mémoire,* `free_0_0` *en libère; les autres sont chargées du contrôle et du calcul. Le successeur de* `estimateFrameNoise_0` *est* `frameController_1` *et n'est donc pas représenté sur ce schéma.*

La figure 3.1 illustre le modèle décrit ci-dessus avec un graphe de tâches représentant un algorithme d'amélioration de la qualité d'image qui applique une réduction de bruit temporelle (*Temporal Noise Reduction*, TNR) à chaque ligne de pixels. Chaque tâche effectuant le même traitement parallèle sur toutes les lignes incluses dans les trames constituant une séquence vidéo[15], le graphe ne comprend qu'un exemplaire de chaque. Les nœuds à simple suffixe (p. ex. `frameController_0`) sont exécutés une fois par trame et ceux à double suffixe (p. ex. `tempUV_0_0`) une fois par ligne; les chiffres indiquent respectivement les numéros de trame et de ligne. Dans le détail, l'application fonctionne comme suit: `hostController` est exécuté par le processeur hôte du SoC pour introduire une trame en mémoire externe; `frameController` lance son trai-

15 En d'autres termes, du point de vue de l'algorithme, les lignes de pixels sont indépendantes.

tement depuis un SWPE; `lineController0` et `1` programment le DMA pour respectivement lire et écrire les données en mémoire externe. La partie critique commence avec `alloc` qui alloue un emplacement en mémoire locale pour une ligne complète. Cet emplacement est rempli par un transfert depuis la mémoire externe orchestré par `src`, et, après traitement (décrit ci-dessous), est renvoyé en mémoire externe par `dst`, après quoi l'emplacement réservé peut être libéré par `free`. `EstimateLineNoise` et `estimateFrameNoise` estiment le niveau de bruit de la trame n afin de calibrer le traitement à appliquer à la trame n+1. Enfin, `spaY`, `tempY`, `TempUV` et `motionDetect` analysent l'image afin de permettre à `fading` d'appliquer la correction adéquate. La figure 3.2 montre l'aspect du graphe de tâches pour le traitement de deux lignes de pixels.

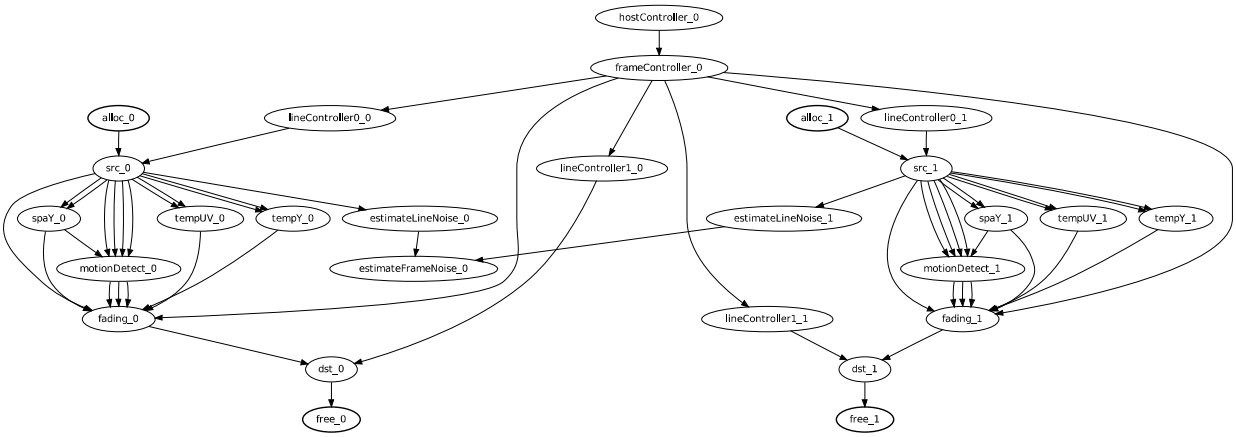


FIGURE 3.2 – *Exemple de graphe de tâches pour l'application TNR incluant le traitement de deux lignes de pixels.*

Il convient de noter que `src` et `dst` ne sont réalisables que par le DMA. Étant donné que la plate-forme ne comprend qu'un seul tel contrôleur, ces tâches sont sérialisées au cours de l'exécution du graphe. Cette disposition assure l'absence d'accès concurrents au niveau du DMA: les transferts sont accomplis un par un.

3.3 Définition du problème

En se fondant sur les modèles présentés ci-dessus, la présente section expose le problème tel qu'il sera abordé dans la suite du chapitre.

3.3.1 Entrées

Soit G = (V, E) un graphe orienté acyclique (*directed acyclic graph*, DAG) modélisant l'application à ordonnancer. Chaque tâche $v_i \in$ V correspond à une invocation d'un acteur et chaque arête (v_i, v_j) $\in$ E représente une dépendance entre deux tâches. On considère un environnement de calcul hétérogène composé de m PE (processeurs généralistes et accélérateurs matériels confondus) tous capables d'accéder à S emplacements dans la mémoire locale. La durée de la tâche v_i sur le PE j est notée $w_{i,j}$. Quand un PE j n'est pas capable d'exécuter la tâche v_i, alors $w_{i,j} = +\infty$. Autrement, étant donné que la durée des tâches peut dépendre des données d'entrée, $w_{i,j}$ est supposé être une variable aléatoire suivant une loi sur $[0, +\infty[$.

Il est de plus nécessaire de distinguer les tâches mémorielles, qui allouent ou libèrent des emplacements en mémoire locale. Leur durée est négligeable mais non nulle. On note $V_M \subset V$ l'ensemble de toutes les tâches mémorielles. Le nombre d'emplacements alloués ou libérés par une tâche $v_i \in V_M$ est cost(v_i), quantité positive quand la tâche alloue des emplacements (tâche *consommatrice*) ou négative quand la tâche en libère (tâche *libératrice*). Chaque tâche consommatrice est appariée avec la tâche libératrice correspondante; il en résulte la bijection appelée pair:

$$\forall v_i \in V_M, \exists! v_j \in V_M, \begin{cases} v_j = \text{pair}(v_i) \in V_M \\ \text{cost}(v_i) + \text{cost}(v_j) = 0 \end{cases}.$$

Enfin, pour cost(v_i) > 0, il existe toujours un chemin de v_i à pair(v_i) dans G pour assurer que la référence de l'emplacement alloué en mémoire est bien passée d'acteur en acteur, depuis sa tâche consommatrice jusqu'à sa libératrice.

3.3.2 Métriques

L'objectif du problème est d'ordonnancer les tâches sur les PE disponibles en conformité avec les contraintes de ressources et les dépendances inter-tâches. Il y a deux métriques à optimiser: l'espérance[16] de la durée totale d'exécution C_{max} et l'utilisation maximale de la mémoire M_{max}. Cette dernière métrique est définie comme suit.

Étant donné un ordonnancement, soit M(t) l'utilisation de la mémoire pour cet ordonnancement à la date t. Par définition:

$$M(t) = \sum_{v_i \in V_M^{\leq}(t)} \text{cost}(v_i)\,,$$

où $V_M^{\leq}(t) \subset V_M$ est l'ensemble des tâches mémorielles ordonnancées jusqu'à l'instant t. D'où:

$$M_{\max} = \max_{t \in [0, C_{\max}]} M(t)\,,$$

et l'ordonnancement doit respecter le nombre d'emplacements disponibles en mémoire:

$$M_{\max} \leq S\,.$$

3.3.3 Discussion

Le problème exposé précédemment est multicritère en cela que l'utilisation de la mémoire et la durée totale d'exécution sont des objectifs contradictoires. Par exemple, la figure 3.3 représente un graphe de tâches où celles indicées par un + allouent exactement un emplacement en mémoire (ce sont des consommatrices), celles indicées par un − en libèrent un, et la tâche i_+ est appariée avec la tâche i_-; de sorte que, quel que soit i, on a: cost(i_+) = +1, cost(i_-) = −1 et i_- = pair(i_+). De plus, la durée des tâches mémorielles est nulle et celle des n autres (c'est-à-dire t_1,...,t_n) est unitaire. Dans ce cas, si l'on ordonnance séquentiellement chaque fil de trois tâches, on atteint M_{max} = 1 mais C_{max} = n, et, si l'on parallélise sur n ressources, on obtient C_{max} = 1 et M_{max} = n.

16 La moyenne est là pour rendre compte des durées de tâche aléatoires.

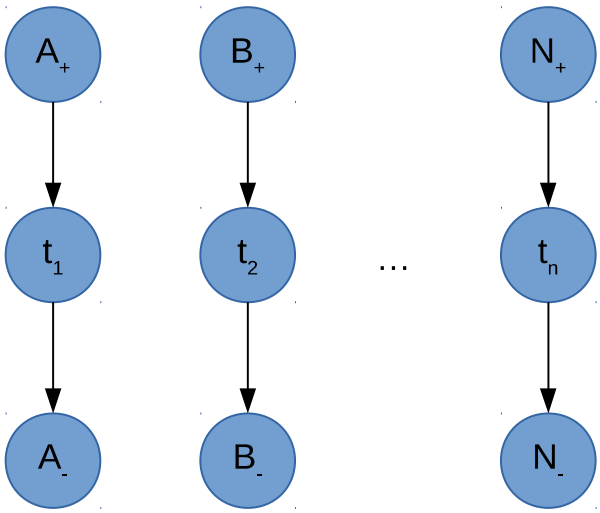
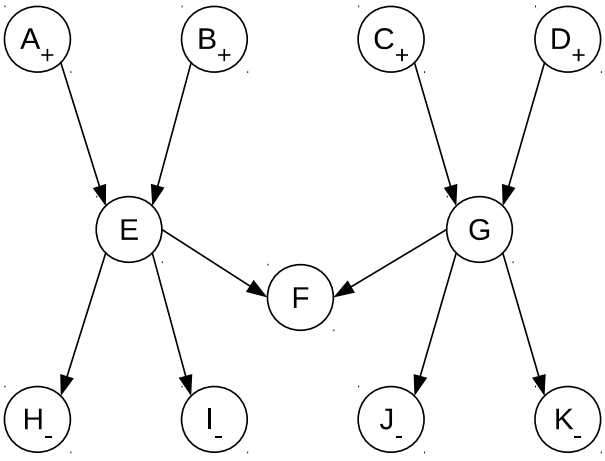


FIGURE 3.3 – *Graphe de tâches menant à des objectifs contradictoires en termes de durée totale d'exécution et de consommation mémorielle. La durée des tâches t_i est 1 sur tous les processeurs.*

3.3.4 Exemple motivant

Toutes les heuristiques d'ordonnancement qui respectent les contraintes d'antériorité ne peuvent pas produire des ordonnancements valides en termes de contraintes mémorielles. En effet, si le nombre d'emplacements disponibles en mémoire est insuffisant, il se peut que l'ordonnanceur aboutisse à une impasse (*deadlock*). Un exemple de graphe de tâches menant à cette situation est présenté à la figure 3.4; les indices ont la même signification que précédemment.



	Original priority
A_+	8
B_+	12
C_+	8
D_+	12
E	6
F	1
G	6
H_-	1
I_-	2
J_-	1
K_-	2

FIGURE 3.4 – *Graphe de tâches aboutissant à une impasse même avec deux emplacements disponibles en mémoire.*

On notera que, dans ce cas, le nombre de machines n'a pas d'incidence quant à l'utilisation de la mémoire. En effet, la mémoire est partagée entre les nœuds et donc son utilisation est seulement influencée par l'ordre dans lequel les emplacements sont alloués et libérés. Suivant les priorités

données par le tableau de la figure 3.4, sur un seul processeur la séquence B_+, D_+ aboutit à une impasse si la mémoire est limitée à deux emplacements : après avoir exécuté B_+ et D_+, les seules tâches prêtes consomment de la mémoire (A_+ ou C_+). Par conséquent, quel que soit le nombre de ressources disponibles, HEFT et SDC échoueront sur cet exemple, dans la mesure où B_+ et D_+ ont des priorités plus fortes que A_+ et C_+. Avec deux emplacements, une solution consiste à exécuter les parties gauche et droite du graphe l'une après l'autre : la séquence A_+, B_+, E, H_-, I_-, C_+, D_+, G, J_-, K_-, F est un ordonnancement valide dans ces conditions. Il en résulte la nécessité d'une heuristique d'ordonnancement prenant en considération les contraintes de mémoire.

3.3.5 NP-difficulté

Il est bien établi que minimiser C_{max} seul est NP-difficile, mais minimiser M_{max} seul l'est aussi. Ce problème est similaire à celui de l'allocation de registres qui est connu pour être NP-difficile [51] par réduction à un problème de coloration de graphe. Cependant, la réduction à notre problème n'est pas triviale. Pour prouver la NP-difficulté, il s'agit de montrer que le problème de décision associé est NP-complet par une réduction à un jeu de galets défini comme suit.

3.3.5.1 Le problème Pebble(K)

En entrée, on dispose d'un graphe de tâches H et d'un ensemble infini de galets marqués. Les galets seront mis sur les nœuds de H. On définit un jeu avec les déplacements autorisés suivants :

1. Enlever un galet d'un nœud, s'il y en a un.
2. S'il y a des galets sur tous les prédécesseurs directs d'un nœud x, alors placer un galet sur x ; ainsi un nœud sans prédécesseur peut toujours recevoir un galet.

Les marqueurs peuvent être des nombres séquentiels utilisés pour compter le nombre de galets mis sur H à chaque instant du jeu en temps constant. Le but du jeu est, en commençant avec un graphe vide de galets, de trouver une séquence de déplacements telle que chaque nœud reçoive exactement un galet. Sethi [49] montre que trouver une séquence de déplacements utilisant moins de K galets est NP-complet[17]. Plus précisément, l'auteur propose un troisième déplacement qui permet de glisser un galet d'un de ses prédécesseurs au nœud v, si tous les prédécesseurs de v ont un galet. Cependant, il est prouvé [57] qu'interdire le glissement augmente toujours d'exactement une unité le nombre de galets nécessaires, quel que soit le graphe H. Il en résulte que les deux versions sont NP-complètes.

3.3.5.2 La minimisation de M_{max} est NP-difficile

Premièrement, il est rappelé que, du fait du modèle de mémoire partagée, le nombre de machines sur lesquelles le graphe d'entrée G est ordonnancé n'a pas d'importance : seul l'ordre dans lequel la mémoire est allouée et libérée compte. Dans l'exemple de la figure 3.4, l'ordonnancement monoprocesseur A_+, B_+, E, H_-, I_-, C_+, D_+, G, J_-, K_-, F utilise deux emplacements en mémoire, tandis que la séquence A_+, C_+, B_+, D_+, E, G, F, H_-, I_-, J_-, K_- en utilise quatre. Deuxièmement, on appelle $M_{max}(M)$ le problème de décision associé à la minimisation de M_{max} : étant donné un entier M et un graphe d'entrée G, y a-t-il un ordonnancement monoprocesseur des tâches tel que $M_{max} \leq M$? Si l'on montre que $M_{max}(M)$ est NP-complet, alors il s'ensuivra que minimiser M_{max} est NP-difficile.

17 Si un nœud peut recevoir plus d'un galet au cours du jeu, alors le problème est PSPACE-complet (et donc NP-difficile), mais probablement pas dans NP [57].

Théorème : $M_{max}(M)$ est NP-complet.

Preuve – Étant donné une entrée de Pebble(K), on construit une solution G = (V, E) de $M_{max}(K)$ comme suit :

- Pour chaque nœud i de H, créer deux sommets i_+ et i_-. On met i_+ dans V_+ et i_- dans V_-.
- L'ensemble des nœuds mémoriels est composé de ceux dans V_+ et V_- seulement : $V_M = V_+ \cup V_-$.
- On apparie ces nœuds : $i_- = \text{pair}(i_+)$.
- Les coûts sont unitaires : $\text{cost}(i_+) = -\text{cost}(i_-) = 1$.
- Tous les nœuds sont mémoriels : $V = V_M$.
- Pour chaque arête (i, j) dans H, on construit une arête $e_{i+j+} = (i_+, j_+)$ et une arête $e_{j+i-} = (j_+, i_-)$. On ajoute e_{i+j+} et e_{j+i-} à E.
- Si i n'a aucun successeur dans H, on construit une arête $e_i = (i_+, i_-)$ et on l'ajoute à E.

Il est clair que cette réduction est polynomiale en la taille de H. La figure 3.6 illustre comment l'entrée de la figure 3.5 se réduit à une entrée de M_{max}. Par exemple, l'arête (a, c) se transforme en deux arêtes : (a_+, c_+) et (c_+, a_-). Comme g n'a pas de successeur, on a seulement une arête (g_+, g_-).

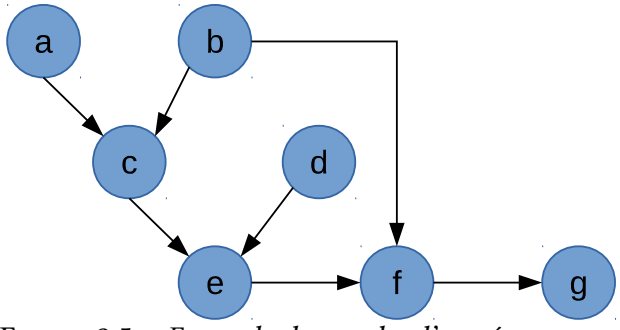


FIGURE 3.5 – *Exemple de graphe d'entrée pour un problème Pebble.*

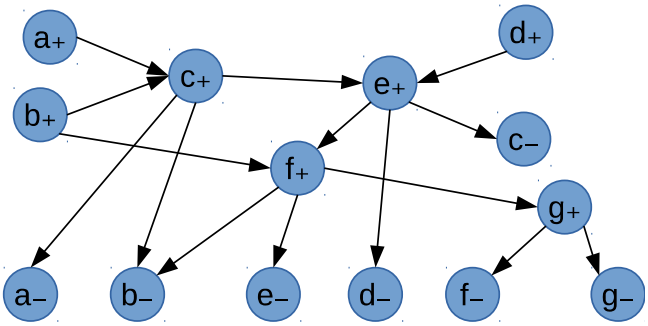


FIGURE 3.6 – *Réduction de l'entrée Pebble ci-contre à une entrée M_{max}.*

Toute solution σ_G de $M_{max}(K)$ est un ordre total $(v_1, \ldots, v_n)$ des sommets de G qui respecte les contraintes d'antériorité. À partir d'une telle solution, il est possible d'en construire une pour Pebble(K). Pour ce faire, on considère les sommets v_i en fonction de l'ordre total de σ_G (de v_1 à v_n). Deux cas se présentent :

1. si $v_i \in V_+$, cela signifie que, d'après la réduction, v_i a la forme i_+ : on place un galet sur le nœud i de H ;
2. si $v_i \in V_-$, cela signifie que, d'après la réduction, v_i a la forme i_- : on enlève le galet du nœud i.

Il s'ensuit que, si $M_{max} = K$, alors le nombre de galets utilisés dans la solution du jeu est K. En effet, par définition, l'utilisation de la mémoire faite par σ_G est le maximum de $M(t)$, ce qui est égal au nombre de sommets de V_+ moins le nombre de sommets de V_- exécutés à l'instant t. Par exemple, la séquence $a_+, b_+, c_+, a_-, d_+, e_+, c_-, d_-, f_+, b_-, e_-, g_+, f_-, g_-$ respecte les contraintes d'antériorité de G et utilise quatre emplacements en mémoire. Elle peut être transformée en temps polynomial en une

solution où tous les nœuds du graphe de la figure 3.5 reçoivent un galet successivement avec un total de quatre galets : on en place un sur le nœud i quand on a i_+ et on l'enlève quand on lit i_-. De plus, toutes les règles du jeu sont respectées (c'est-à-dire que la solution est correcte) :

- tous les nœuds recevront un galet une et une seule fois puisque les nœuds de V_+ sont exécutés une fois chacun dans l'ordonnancement de G ;
- tous les nœuds sans prédécesseur peuvent recevoir un galet à tout instant ;
- si un nœud a un prédécesseur, alors ses prédécesseurs perdront leur galet seulement une fois que celui-ci aura reçu le sien. : si i est un prédécesseur de j dans H, alors i_- est un successeur de j_+ dans G, et donc le retrait du galet de i (l'exécution de i_-) ne peut se produire qu'après la pose du galet de j (l'exécution de j_+), puisque σ_G respecte l'ordre topologique ;
- il est correct de prélever un galet d'un nœud i dans H car ce galet a forcément été posé auparavant : i_+ est toujours un prédécesseur de i_- dans G.

De ce qui précède, il résulte que s'il est possible de résoudre $M_{max}(K)$ en temps polynomial alors il est également possible de résoudre Pebble(K) en temps polynomial. ∎

3.4 Description de la solution proposée

Cette section décrit la solution proposée, qui consiste essentiellement à modifier les priorités utilisées dans les heuristiques d'ordonnancement de liste. Dans un premier temps, des définitions et propositions utiles par la suite sont présentées, suivies de la description de la méthode d'ajustement des priorités proposée. Est également introduite une modification de l'algorithme d'insertion typiquement employé dans les ordonnanceurs de liste, afin de faire face aux contraintes mémorielles. Enfin, la dernière partie décrit plus avant la stratégie d'ordonnancement auto-séquencé utilisée lors des expérimentations.

Définition 1 : Un *ensemble mémoriel* est un ensemble de nœuds d'un graphe de tâches qui comprend tous les chemins depuis une tâche consommatrice jusqu'à sa libératrice appariée, y compris celles-ci. Certains de ces ensembles sont combinés de façon à former des *groupes mémoriels* tels qu'un groupe mémoriel comprenne tous les ensembles mémoriels ayant au moins un nœud en commun.

Suite à cette définition, il vient qu'un ensemble mémoriel qui n'a de sommet en commun avec aucun de ses homologues est aussi un groupe mémoriel à lui seul. Par exemple, dans le cas du graphe de la figure 3.1, le groupe correspondant au traitement de la ligne 0 de la trame 0 est constitué par `alloc_0_0`, `free_0_0` et les neuf tâches situées entre elles. De plus, la figure 3.13 montre un graphe plus complexe avec plusieurs groupes.

Définition 2 : Étant donné deux groupes mémoriels A et B, A est un *ancêtre* de B s'il existe un chemin d'un nœud v_A dans A à un nœud v_B dans B.

Définition 3 : La *borne inférieure réalisable* (BIR) du coût mémoriel est le maximum du nombre total d'emplacements consommés par un groupe, parmi tous les groupes.

On en déduit deux conditions permettant de réaliser cette borne inférieure :

– C1 : Les ensembles formés des priorités des tâches consommatrices appartenant à différents groupes ne se chevauchent pas.

– C2 : Les tâches consommatrices appartenant à des ancêtres ont des priorités supérieures[18] à celles de leurs descendantes.

L'idée est que C1 traite le cas des groupes indépendants, tandis que C2 aborde celui des groupes dépendants, comme l'illustre la figure 3.7.

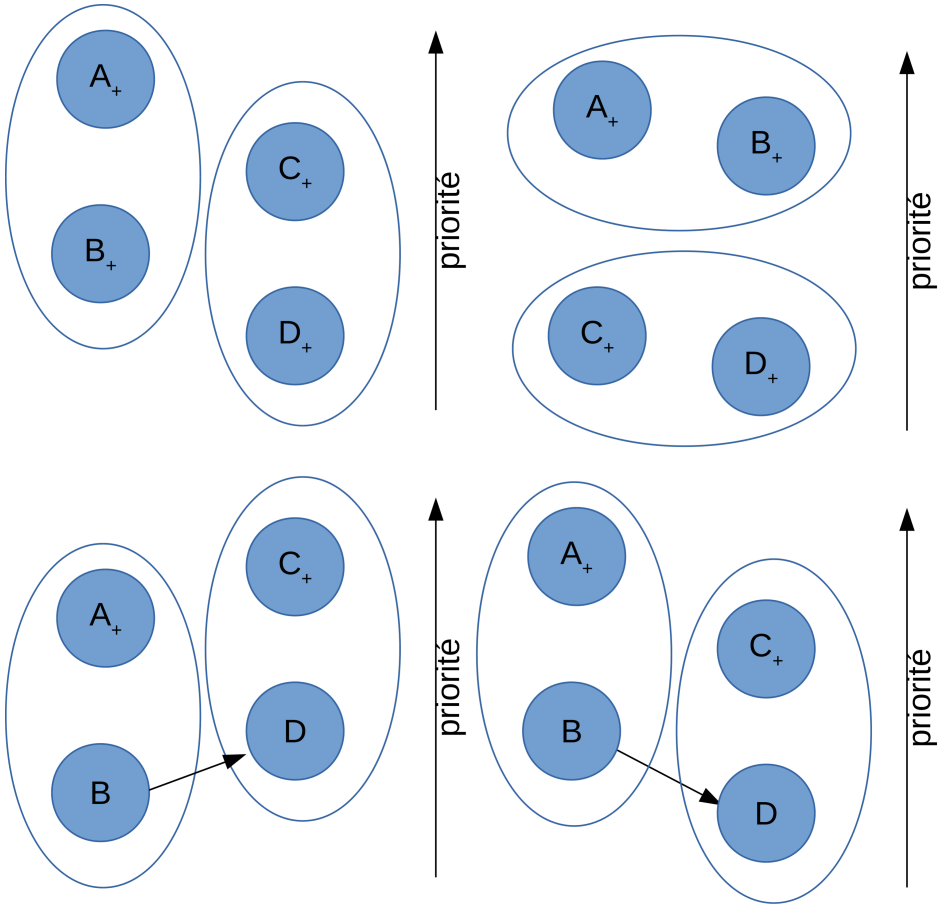


FIGURE 3.7 – *Les quatre sous-figures donnent des exemples de respect et de violation des conditions C1 et C2. Les nœuds situés plus haut ont des priorités supérieures. Les ellipses représentent les groupes mémoriels. Les deux sous-figures du haut illustrent la condition C1, celles du bas la condition C2; celles de gauche les enfreignent, tandis que celles de droite les respectent.*

Proposition : Les conditions C1 et C2 sont suffisantes pour ordonnancer selon le BIR.

Preuve – Premièrement, on considère deux groupes déconnectés A et B, c'est-à-dire qu'il n'existe pas de chemin entre un nœud de A et un nœud de B. Soit $P : V \longrightarrow \mathbb{N}$ l'application qui transforme un nœud en sa priorité. Alors la condition C1 garantit que :

$$\forall (v_A, v_B) \in A \times B, \begin{cases} \exists (v'_A, v'_B) \in A \times B, P(v'_A) > P(v'_B) \implies P(v_A) > P(v_B) \\ \exists (v'_A, v'_B) \in A \times B, P(v'_A) < P(v'_B) \implies P(v_A) < P(v_B) \end{cases}.$$

18 Pour rappel, plus la priorité d'une tâche est élevée, plus tôt celle-ci sera ordonnancée.

En termes d'ordonnancement, cela signifie que, si une tâche consommatrice de A (respectivement B) est ordonnancée en premier, alors toutes les tâches consommatrices de A seront ordonnancées avant celles de B (respectivement A), ce qui assurera l'absence d'impasse due à un manque de mémoire. Par exemple, dans le cas du graphe de la figure 3.4, cela assure que A_+ et B_+ seront ordonnancées ensemble, avant C_+ et D_+, ou l'inverse, et donc la totalité du groupe sera ordonnançable.

On suppose maintenant qu'un nœud de B a une dépendance d'entrée vis-à-vis d'un nœud de A, ce qui fait de A un ancêtre de B. On note A^c (respectivement B^c) l'ensemble des nœuds de A (respectivement B) qui consomment de la mémoire. Alors la condition C2 exige que :

$$\forall (v_A, v_B) \in A^C \times B^C, P(v_A) > P(v_B)\ .$$

Ainsi les tâches consommatrices de A seront ordonnancées en premier. Il en résulte que la dépendance sera satisfaite quand celles de B seront ordonnancées, évitant par là même le gaspillage de mémoire. ∎

Afin de remplir ces conditions, le processus d'ordonnancement doit être adapté puisque le simple comptage des emplacements disponibles en mémoire introduit des dépendances implicites qui n'apparaissent pas dans le graphe initial et, par conséquent, ne peuvent pas être prises en compte par les ordonnanceurs traditionnels. Pour y remédier, on construit un nouveau graphe de tâches :

Définition 4: Le graphe d'indépendance associé à une application est un graphe non orienté dont les sommets représentent seulement les tâches mémorielles. Les arêtes sont telles que deux nœuds sont connectés si et seulement si il n'existe pas de chemin entre eux dans le graphe de tâches original.

L'idée est de rendre compte des relations d'antériorité liées aux contraintes de mémoire entre tâches mémorielles qui n'apparaissent pas comme des dépendances de données. Le recours à ce graphe permet un ajustement de priorités visant à avancer l'exécution des tâches libératrices, sachant qu'elles constituent le principal goulet d'étranglement de l'ordonnancement.

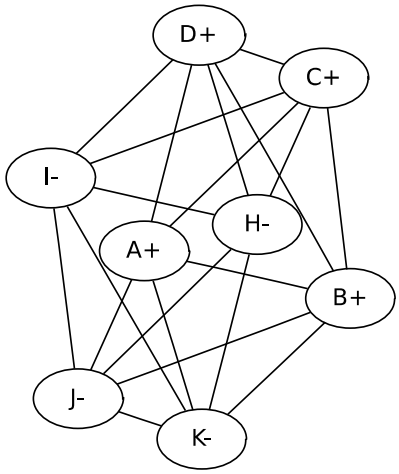


FIGURE 3.8 – *Graphe d'indépendance correspondant au graphe de la figure 3.4.*

La figure 3.8 illustre ce à quoi ressemble un graphe d'indépendance à partir du graphe de tâches de la figure 3.4. Soient les paires de tâches consommatrices-libératrices suivantes: (A_+, H_-), (B_+, I_-), (C_+, J_-) et (D_+, K_-). Le graphe de tâches original comprend deux groupes mémoriels: C_0 = {A_+, B_+, E, H_-, I_-} et C_1 = {C_+, D_+, G, J_-, K_-}. Dans C_0, il existe des chemins de A_+ et B_+ à H_- et I_-. De même, dans C_1, J_- et K_- sont tous deux accessibles depuis C_+ et D_+. Tous les autres nœuds mémoriels sont déconnectés dans le graphe original, et donc adjacents dans le graphe d'indépendance.

3.4.1 Ajustement des priorités

La présente section introduit un mécanisme d'ajustement des priorités applicable en présence de contraintes mémorielles aux algorithmes d'ordonnancement de liste à priorités statiques.

Chaque tâche libératrice v_r recevra un bonus de priorité P_B équivalent au total des priorités de l'ensemble V_C^* de tâches v_c satisfaisant aux critères suivants:

1. v_c est adjacente à v_r dans le graphe d'indépendance;
2. cost(v_c) > 0, c'est-à-dire que v_c est une tâche consommatrice;
3. l'une des deux assertions suivantes est vraie:
   (a) P(v_c) < P(pair(v_r)),
   (b) pair(v_c) n'est pas adjacent à pair(v_r) dans le graphe d'indépendance.

$$P_B(v_r) = \sum_{v_c \in V_C^*} P(v_c)$$

Ce cadre formel peut se concevoir plus intuitivement en termes de cycles de vie.

Définition 5: Un *cycle de vie mémoriel* est une portion d'ordonnancement s'étendant de la date de début d'une tâche consommatrice à la date de fin de sa libératrice appariée.

Le raisonnement qui sous-tend l'ajustement des priorités est ainsi d'empêcher les cycles de vie de se chevaucher de sorte à limiter l'empreinte mémorielle globale.

Le critère 1 assure que seules les tâches sans relations d'antériorité préexistantes sont considérées, afin d'éviter une boucle de propagation de bonus. Le critère 2 empêche les tâches libératrices de s'influencer mutuellement. Les critères 3a et 3b visent respectivement à remplir les conditions C1 et C2. Plus précisément, le critère 3a tend à prévenir le chevauchement des cycles de vie mémoriels en calculant les bonus des tâches libératrices provenant de groupes dont les consommatrices ont des priorités supérieures à partir des consommatrices ayant des priorités inférieures; mais cela peut parfois s'avérer insuffisant, comme le montre la section 3.4.2. Enfin, le critère 3b signifie qu'il y a un chemin dans le graphe de tâches original depuis un consommateur dans le groupe obtenant le bonus jusqu'à un libérateur dans celui donnant le bonus, pour assurer que les tâches en amont ont toujours des priorités supérieures. Ces priorités ajustées sont ensuite propagées au reste du graphe via une seconde passe de la phase normale d'assignation de priorités.

Pour illustrer les exigences susmentionnées, on considère le graphe d'indépendance de la figure 3.8 et l'on suppose que H_- est candidat à un ajustement de priorités en tant que nœud libérateur. Les tâches suivantes sont adjacentes à H_- dans le graphe d'indépendance et donc satisfont

au premier critère : C_+, D_+, I_-, J_- et K_-. Parmi celles-ci, seules les consommatrices remplissent le deuxième critère : C_+ et D_+. Ensuite, $P(C_+) = P(A_+)$ et $P(D_+) > P(A_+)$, et A_+ est adjacente à J_- et K_- dans le graphe d'indépendance, donc H_- n'aura pas de bonus de priorité. De même, si I_- est candidat, alors C_+ et D_+ satisfont les deux premiers critères. D'autre part, $P(C_+) < P(B_+)$ donc C_+ remplit aussi le critère 3a (mais pas 3b). Il résulte de ceci que I_- aura un bonus égal à $P(C_+) = 8$. L'ajustement des priorités des deux autres tâches libératrices peut être déduite *via* un raisonnement analogue. Après propagation des bonus, on obtient les priorités présentées dans le tableau 4.

Nœud	Priorité originale	Bonus	Priorité ajustée	Priorité forcée
A_+	8	8	16	21
B_+	12	8	20	21
C_+	8	8	16	16
D_+	12	8	20	20
E	6	8	14	14
F	1	0	1	1
G	6	8	14	14
H_-	1	0	1	1
I_-	2	8	10	10
J_-	1	0	1	1
K_-	2	8	10	10

TABLEAU 4 – *Priorités avant et après ajustement dans la figure 3.4.*

3.4.2 Forçage des priorités

Dans certains rares cas de figure, la méthode d'ajustement des priorités présentée n'est pas suffisante pour atteindre la condition C1. La figure 3.4 offre un exemple d'une telle situation. Le graphe comprend deux groupes déconnectés C_0 et C_1, et le BIR est 2. Comme il a été vu à la section 3.4.1, la priorité de I_- doit recevoir un bonus de C_+ et celle de K_- un bonus de A_+, ce qui aboutit aux valeurs ajustées du tableau 4. À ce stade, du fait de la symétrie du graphe, A_+ a la même priorité que C_+ et B_+ la même que D_+. Par suite, B_+ et D_+ devraient être ordonnancées avant A_+ et C_+, ce qui causerait l'utilisation d'au moins trois emplacements en mémoire au lieu des deux disponibles. D'où la nécessité de forcer les priorités à des valeurs telles que A_+ et B_+ soient ordonnancées ensemble avant C_+ et D_+ : la plus petite priorité de l'un de ces deux groupes doit être supérieure à la plus grande de l'autre groupe.

Pour garantir le forçage des priorités, on a recours à l'algorithme 1 qui applique la condition C1 directement. Le raisonnement qui sous-tend cet algorithme est que les priorités de certaines tâches consommatrices peuvent nécessiter une augmentation afin d'éviter des chevauchements entre groupes. Pour ce faire, la liste de priorités est parcourue en sens inverse, c'est-à-dire par ordre de priorité croissante, et, à chaque recouvrement détecté, la priorité du consommateur au rang le plus bas est augmentée. Grâce à ce mécanisme, le forçage des priorités n'altère pas les tâches déjà parcourues et l'algorithme ne nécessite qu'une seule passe.

Compter le nombre de consommateurs dans chaque groupe ;
// Traverser la liste de priorités en sens inverse en considérant uniquement les tâches consommatrices
pour chaque *nouveau groupe C traversé* **faire**
    si *toutes les tâches dans le groupe précédent C′ n'ont pas encore été traversées* **alors**
        Trouver la tâche T′ de C′ avec la plus haute priorité ;
        tant qu'*il y a des tâches T dans C telles que P(T) ≤ P(T′)* **faire**
            // Élever la priorité de la tâche T
            P(T) ← P(T′) + 1
        fin
    fin
fin

ALGORITHME 1 – *Forçage de priorités.*

En pratique, le bonus supplémentaire introduit par cet algorithme est très faible: l'ajustement initial des priorités est déjà très efficace. Par exemple, dans le cas de la figure 3.4, puisque les deux groupes mémoriels ont des cycles de vie qui se chevauchent, soit les priorités de A_+ et B_+ soit celles de C_+ et D_+ doivent être forcées. On suppose que, du fait de l'ordre topologique, C_0 est le dernier groupe à être traversé par l'algorithme, et donc les priorités de A_+ et B_+ seront affectées. Sachant que la priorité ajustée la plus élevée de C_1 est $P(D_+) = 20$, A_+ et B_+ auront donc tous deux leur priorité élevée à 20 + 1 = 21, comme le montre le tableau 4. Ainsi, tandis que HEFT et SDC mènent invariablement à une impasse avec seulement deux emplacements disponibles en mémoire et les priorités originales, le mécanisme proposé assure que ces nouvelles priorités permettent d'aboutir à un ordonnancement valide.

Enfin, bien que cet algorithme soit attrayant par sa simplicité et par les garanties qu'il offre, il ne doit être utilisé qu'en dernier recours et en combinaison avec la méthode d'ajustement des priorités présentée précédemment. En effet, seul il n'est pas suffisant pour réaliser la condition C2 et ne permet donc pas de se prémunir contre tous les risques d'impasse. De plus, appliqué sans ajustement préalable, il produit un « tassement » des priorités qui nuit à la qualité de l'ordonnancement – en termes de durée totale d'exécution – en annihilant l'effet de pipeline souhaitable dans la mesure où il n'enfreint pas les contraintes mémorielles. Ces deux points sont illustrés par les expériences de la section 3.5.2.

3.4.3 Gestion de l'insertion

Bon nombre d'heuristiques autorisent l'ordonnancement de tâches dans des créneaux d'inactivité. Cette section montre comment adapter de tels mécanismes aux contraintes de mémoire telles qu'elles sont appréhendées dans le modèle proposé. À cet égard, le point critique est l'insertion des tâches mémorielles, en particulier les consommatrices, qui risque de bloquer l'exécution de tâches postérieures déjà ordonnancées, en cas de chevauchement des cycles de vie.

Soit $s(t)$ le nombre d'emplacements disponibles en mémoire à l'instant t et $V_M^{=}(t)$ l'ensemble de toutes les tâches mémorielles en cours d'exécution à l'instant t; $s(t)$ représente l'état de la mémoire locale à chaque étape du processus d'ordonnancement et est supposé connaissable rétrospectivement pour toute étape $t_0 < t$ *via* la formule de récurrence suivante:

$$\forall t \geq t_0, s(t) = s(t_0) + \sum_{v_m \in \bigcup_{t' \in [t_0,t]} V_M^{=}(t')} \text{cost}(v_m) \,.$$

Soit I(t) l'ensemble des créneaux disponibles à l'instant t. Pour tout $i \in$ I(t), on définit start(i) la date de début de disponibilité de i et end(i) sa date de fin, et on en déduit la durée de i : d(i) = end(i) − start(i). Pour une tâche v que l'on souhaite insérer, on note EST(v) et $\tilde{d}(v)$ la date de début d'exécution estimée et la durée de v, respectivement. Alors, une tâche consommatrice v_c peut être insérée dans un créneau i donné si les assertions suivantes sont vraies :

– le créneau considéré a une durée suffisante :

$$d(i) \geq \tilde{d}(v_c);$$

– il y a suffisamment de mémoire disponible au point d'insertion :

$$\exists (t_0, t_0') \in [\text{start}(i), \text{end}(i)]^2, \begin{cases} \forall t \in [t_0, t_0'], s(t) \geq \text{cost}(v_c) \\ t_0 \leq \text{EST}(v_c) \leq t_0' \end{cases};$$

– l'insertion n'affectera pas les tâches suivantes :

$$\forall t \geq \text{EST}(v_c), s(\text{EST}(v_c)) + \text{cost}(v_c) + \sum_{v_m \in \bigcup_{t' \in [\text{EST}(v_c),t]} V_M^{=}(t')} \text{cost}(v_m) \geq 0 \,.$$

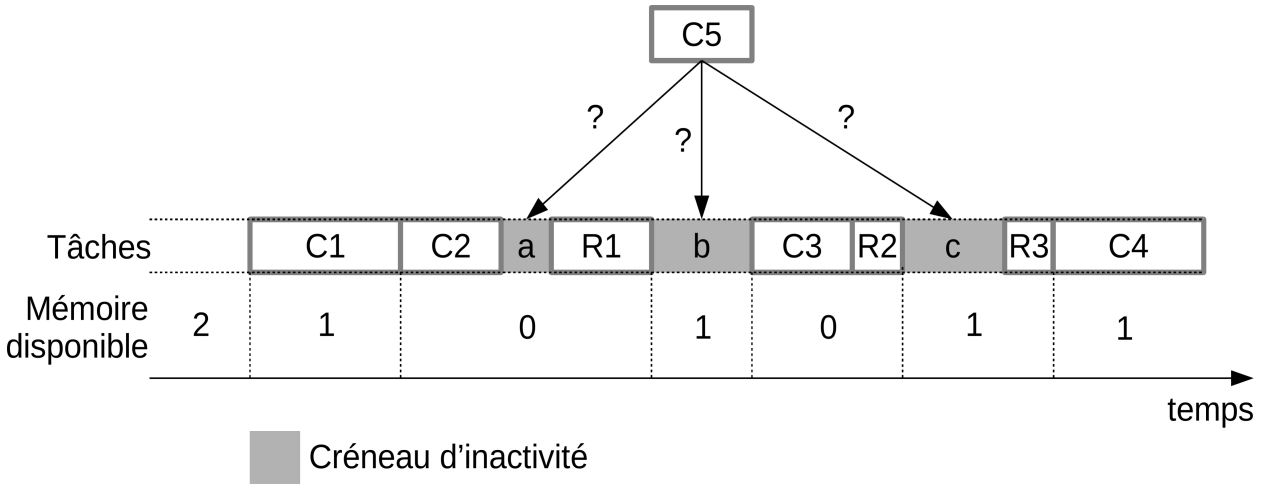


FIGURE 3.9 – *Tentative d'insertion d'une tâche consommatrice dans un créneau d'inactivité. Seules les tâches mémorielles sont représentées : C1 à C5 représentent les consommatrices et R1 à R3 les libératrices. Les créneaux sont notés a, b et c.*

La figure 3.9 illustre comment l'insertion fonctionne en présence de tâches mémorielles. On suppose que le consommateur C5 est candidat à l'insertion et que trois créneaux d'inactivité s'offrent potentiellement à lui. Le premier critère indique que C5 ne tient pas dans le créneau a car $d(a) < \tilde{d}(C5)$. Le deuxième critère autorise C5 à être inséré dans b ou c puisque les deux disposent d'un emplacement disponible en mémoire. Selon le dernier critère, l'insertion d'une tâche ne doit jamais rendre le nombre d'emplacements disponibles négatif, y compris dans le futur. Pour appliquer cette règle, ce nombre doit être recalculé à tous les instants postérieurs au point d'insertion. Dans cet exemple, insérer C5 dans le créneau b aurait pour effet d'empêcher l'exécution de C3 par manque de mémoire, ce n'est donc pas autorisé. Finalement, la seule solution est d'ordonnancer C5 dans le créneau c.

3.4.4 Ordonnancement auto-séquencé

Pour faire face à la variabilité de la durée des tâches, l'heuristique d'ordonnancement a été modifiée comme suit. Premièrement, on calcule la priorité et un ordonnancement statique de chaque tâche en se fondant sur l'espérance de la variable aléatoire $w_{i,j}$ qui donne la durée de la tâche i sur le processeur j. Ensuite, au moment de l'exécution, on utilise l'ordonnancement pré-calculé pour assigner et ordonner les tâches, c'est-à-dire que chacune d'entre elles est exécutée sur le même processeur et dans le même ordre conformément au calcul de l'ordonnanceur. Cependant, étant donné que la durée des tâches peuvt diverger de la valeur moyenne sur laquelle l'ordonnanceur s'est basé, les dates de début d'exécution peuvent elles aussi différer. De ce fait, une tâche est exécutée, non pas à l'instant prévu par l'ordonnanceur, mais dès que ses dépendances (dans le graphe de tâches) sont satisfaites et son prédécesseur (sur le processeur assigné) a rendu la main. Il s'agit d'une stratégie d'ordonnancement auto-séquencé telle que décrite à la section 3.1 et dont la justification sera exposée à la section 3.5.1.

3.5 Expérimentations

Les méthodes d'ajustement des priorités et d'insertion des tâches mémorielles présentées à la section précédente ont été incorporées dans HEFT et SDC. Il convient de noter que ces deux techniques sont compatibles avec tout algorithme d'ordonnancement de liste à priorités statiques. Les expériences ont été menées sur deux applications réelles : TNR, présenté à la section 3.2.3, et l'algorithme de décodage vidéo H.264. Comme aucune plateforme matérielle n'était disponible, les ordonnancements ont seulement été simulés de la façon décrite à la section 3.4.4, sans les exécuter réellement.

Tous les essais ont consisté à comparer les durées totales d'exécution des ordonnancements obtenus après ajustement des priorités à celles de la version non ajustée ; l'insertion est toujours activée. Les résultats sont des moyennes sur mille exécutions avec des durées de tâche aléatoires calculées de la manière suivante :

1. On définit la durée de référence w_{r} pour chaque acteur :
   (a) Pour chaque type d'acteur (`src`, `fading`, etc.), on définit une durée unitaire par nombre de pixels.
   (b) On définit la durée de référence w_{r} pour chaque acteur en multipliant la durée unitaire par le nombre de pixels traités (ligne ou macrobloc).
2. On crée différentes instances de graphes de tâches où les durées varient autour de cette valeur de référence.
   (a) Afin d'obtenir des variations similaires pour les différentes instances d'une même tâche, on définit d'abord la durée aléatoire moyenne $\bar{w}$ de cet acteur en choisissant un facteur de dispersion $a \geq 1$ tel que $\bar{w} \in [\frac{w^r}{a}, aw^r]$. Pour ce faire, on a recours à la loi bêta qui a un support sur [0, 1], et dont l'espérance est 0,5 quand $\alpha = \beta$. Dans le cas présent, on choisit $(\alpha, \beta) = (2, 2)$:

$$\bar{w} = w^r \left(\mathrm{Beta}(2,2)(a - 1/a) \; + 1/a\right) .$$

De plus, on impose :

$$\forall i, a \leq \sqrt{w^r_{i,j_s}/w^r_{i,j_h}} ,$$

où w^r_{i,j_s} et w^r_{i,j_h} sont les durées de référence de la tâche v_i respectivement sur un SWPE et un HWPE. Ceci garantit qu'un SWPE ne sera jamais plus rapide qu'un HWPE.

(b) La durée finale de chaque instance de tâche est calculée de façon analogue avec le même facteur de dispersion *a*.

3.5.1 TNR

Dans un premier temps, les heuristiques sont alimentées avec un graphe de tâches décrivant le traitement de 10 lignes de 1000 pixels chacune. Étant donné que cet exemple ne présente aucun risque d'impasse puisque sa BIR vaut 1, l'algorithme de forçage des priorités n'est pas utilisé. La plateforme simulée est composée d'un DMA, un SWPE et cinq HWPE (un par acteur de calcul accéléré). La figure 3.10 illustre les résultats.

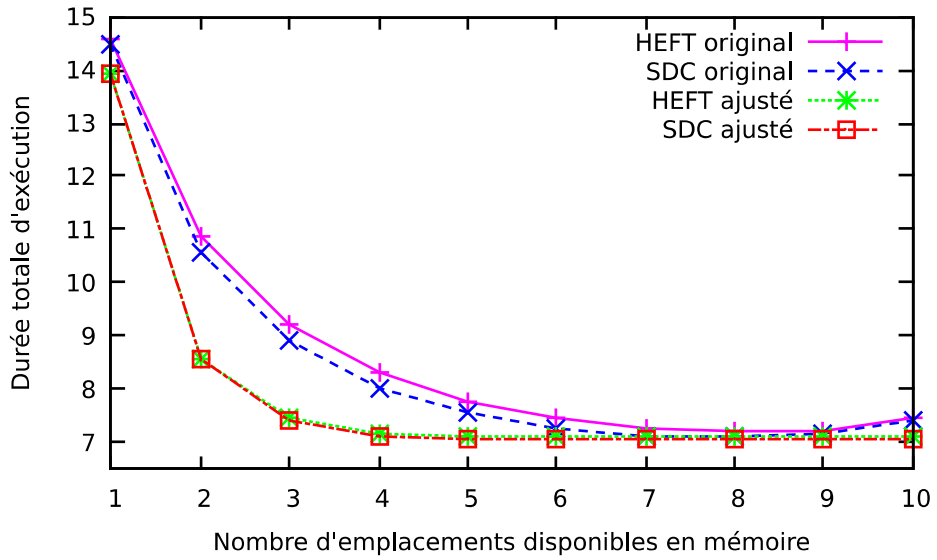


FIGURE 3.10 – *TNR avec 10 lignes de 1000 pixels. Durée totale d'exécution en fonction de la capacité mémorielle.*

Les ordonnancements obtenus sont toujours meilleurs avec ajustement que sans ; cela est vrai aussi bien pour HEFT que pour SDC. Les accélérations relevées vont de 4 % pour un emplacement à 20 % pour deux, avec une moyenne à 10,6 %. La faible accélération pour un emplacement s'explique par le potentiel de recouvrement limité : seule une ligne peut être traitée à la fois. Inversement, l'accélération élevée pour deux emplacements est due aux décisions médiocres prises par la version non ajustée, qui tente d'ordonnancer tous les consommateurs en même temps puisqu'ils ont la même priorité. Néanmoins, cet écart disparaît lorsque le nombre d'emplacements augmente.

Dans une deuxième série d'expériences, la capacité mémorielle est fixée à deux emplacements et le nombre de lignes de pixels en entrée varie de 10 à 100, tous les autres paramètres demeurant inchangés. Cela permet d'évaluer l'impact de la taille de l'application sur la durée d'exécution, comme le représente la figure 3.11.

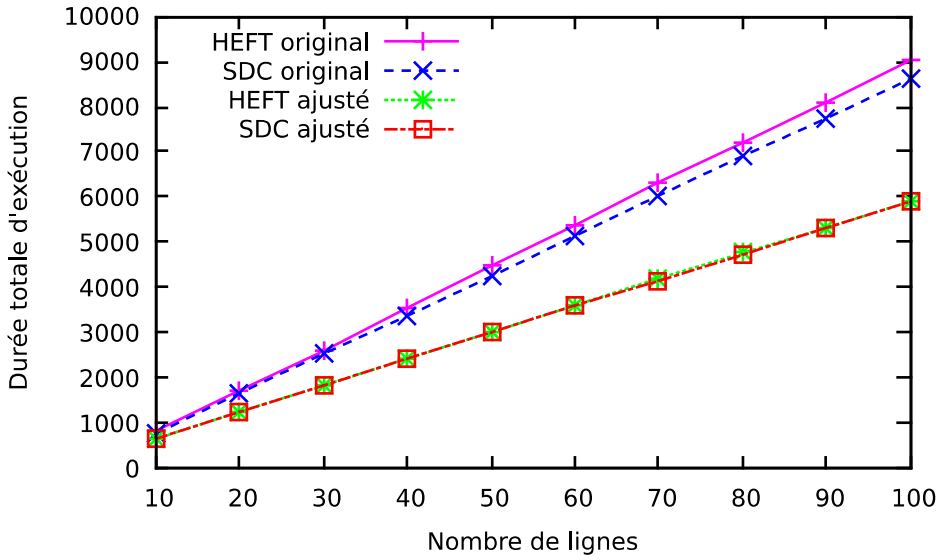


FIGURE 3.11 – *TNR avec deux emplacements en mémoire. Durée totale d'exécution en fonction du nombre de lignes.*

Les résultats montrent que la durée d'exécution s'accroit linéairement avec le nombre de lignes de pixels et que la pente est de l'ordre de 20 % moindre dans le cas ajusté, comme prévu, à la fois pour HEFT et SDC.

Enfin, une méthodologie visant à évaluer les bénéfices d'un ordonnancement auto-séquencé – c'est-à-dire si la décision concernant l'ordre des tâches devrait ou non être laissée à l'exécution – est présentée. À cette fin, trois types d'ordonnancements sont considérés :

- un ordonnancement de *référence*, totalement statique, basé sur les durées de tâche de référence telles que décrites au début de la présente section[19] ;
- un ordonnancement *auto-séquencé* défini de la façon décrite à la section 3.4.4, qui modélise l'exécution d'une application telle que :
  - les durées de tâche exactes ne sont connues qu'à l'exécution ;
  - les durées de tâche moyennes sont connues à la compilation ;
- un ordonnancement dit « *oracle* », connaissant toutes les durées de tâche exactes à la compilation.

La différence entre les durées d'exécution des deux derniers ordonnancements permet de mesurer le gain potentiel qu'apporterait un ordonnanceur partiellement dynamique dans le cas d'une exécution réelle. Ainsi, le résultat de l'oracle peut être vu comme une borne inférieure sur la durée totale d'exécution. Cette méthodologie a été appliquée à l'application TNR, avec les mêmes paramètres que précédemment, pour comparer les durées d'exécution obtenues avec soit un ordonnanceur auto-séquencé soit un oracle en fonction du facteur de dispersion. La figure 3.12 illustre ces résultats.

19 Le seul objet de cet ordonnancement de référence est de servir de fondements pour construire l'ordonnancement auto-séquencé. Ainsi, les durées d'exécution qui en résultent ne sont pas significatives dans le cadre de cette étude et ne sont donc pas représentées.

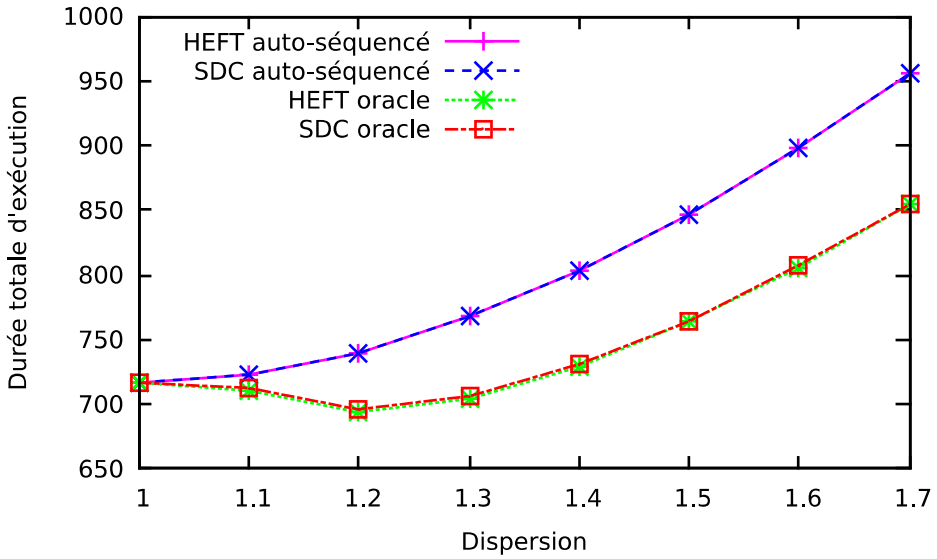


FIGURE 3.12 – *TNR avec 10 lignes de 100 pixels. Durée totale d'exécution en fonction du facteur de dispersion.*

On observe que l'écart s'accroit avec la variation de la durée des tâches, jusqu'à atteindre 10,5 %. Cela est cohérent avec la capacité de l'oracle à compenser ces variations en réassignant une tâche chronophage à un processeur différent, ou en tirant parti des créneaux d'inactivité pour l'insérer. Dans ce cas, on en conclut qu'il serait rentable d'introduire un ajustement dynamique de l'ordonnancement si les variations observées avec l'application correspondent à un facteur de dispersion supérieur à 1,2.

3.5.2 H.264[20]

Pour ce jeu d'expérimentations, un modèle simplifié[21] de décodeur H.264, représenté par la figure 3.13, est utilisé. Dans ce modèle, chaque emploi d'un macrobloc, que ce soit en tant que référence ou pendant son décodage, doit être précédé d'une allocation en mémoire, modélisée par une tâche consommatrice dans le graphe de tâches, et suivie d'une libération. Pour des raisons de simplicité, les macroblocs ne sont pas mis en cache, d'où la nécessité de systématiquement recharger ceux requis pour les calculs. De ce fait, les tâches chargées du traitement des macroblocs suivants – dans l'ordre de balayage – ont des dépendances de données vis-à-vis de celles affectées aux précédents. Contrairement au TNR, il n'est pas possible d'ordonnancer le décodeur H.264 avec une capacité mémorielle arbitrairement basse puisque certaines tâches nécessitent jusqu'à quatre macroblocs simultanément: la BIR est donc 4. La plateforme simulée est composée d'un DMA, un SWPE et quatre HWPE (un par acteur de calcul accéléré).

20 Pour une description de principe de H.264, se reporter au chapitre 1.
21 L'interprédiction n'est pas considérée.

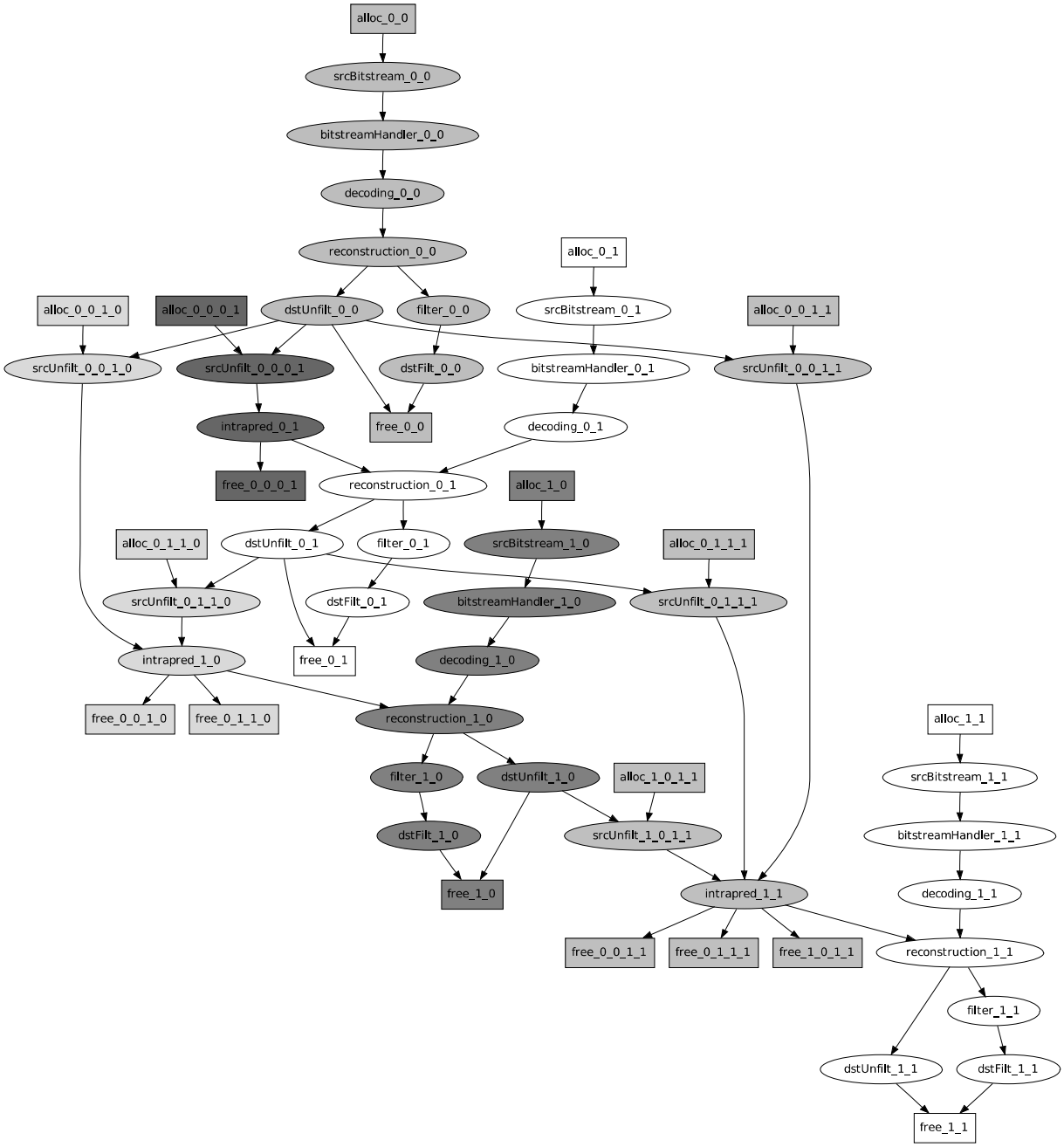


FIGURE 3.13 – *Graphe de tâches d'un décodeur H.264 pour quatre macroblocs dépendants. Les sept groupes mémoriels sont représentés en différentes nuances de gris et blanc. Les nœuds consommateurs et libérateurs apparaissent dans des boites carrées.*

Les ordonnancements avec ajustement des priorités ne surpassent plus leurs homologues non ajustés, au contraire. Cela résulte de l'ajustement des priorités qui tend à empêcher le recouvrement des groupes. Pour pallier ce défaut, un facteur de bonus $\mathrm{BF} \in [0, 1]$ est utilisé pour atténuer l'ajustement des priorités : $\forall v \in \mathrm{V}$, $\mathrm{P}_{\mathrm{adjusted}}(v) = \mathrm{P}_{\mathrm{original}}(v) + \mathrm{P}_{\mathrm{B}}(v) \times \mathrm{BF}$. Dans le premier jeu de simulations, les ordonnanceurs ont reçu un graphe de tâches décrivant le traitement de trois lignes de trois macroblocs. Les figures 3.14 et 3.15 illustrent ces résultats.

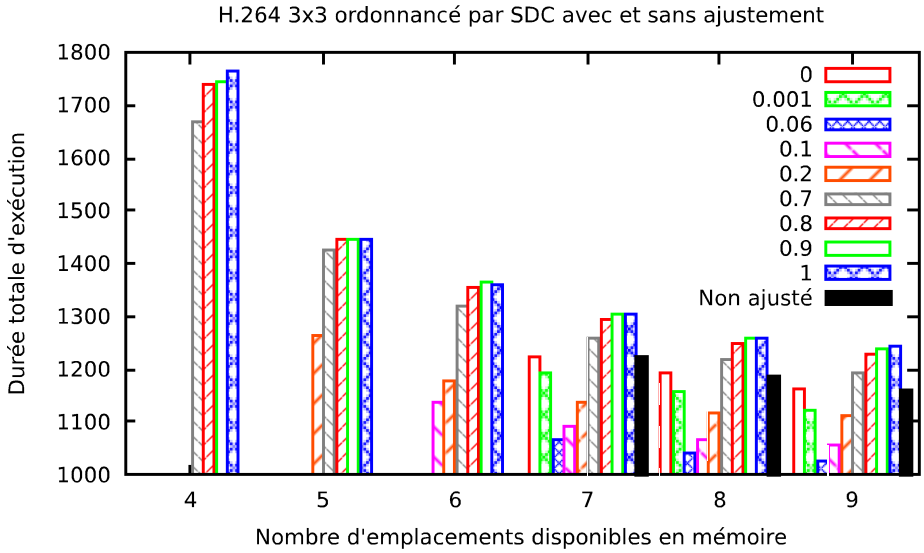


FIGURE 3.14 – *H.264 avec 3×3 macroblocs ordonnancé par SDC.*

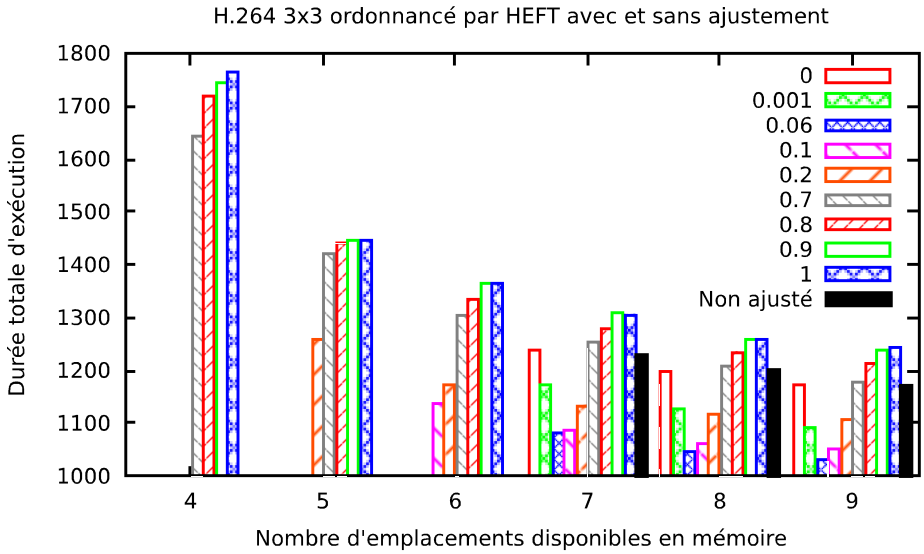


FIGURE 3.15 – *H.264 avec 3×3 macroblocs ordonnancé par HEFT.*

L'absence de barre dans le diagramme signifie qu'aucun ordonnancement licite n'a pu être produit par manque de mémoire. Il peut être observé que plus le facteur de bonus diminue, plus l'espace en mémoire nécessaire pour obtenir des ordonnancements valides augmente. Cela s'explique par le fait qu'un facteur de bonus bas rapproche les priorités ajustées de leur valeur originale. En particulier, avec un facteur de bonus nul, seul le forçage des priorités est effectif; ce qui implique que la condition C2 ne soit pas systématiquement remplie, d'où l'absence de solution pour les nombres d'emplacements les plus faibles. Pour une capacité mémorielle inférieure à sept emplacements, les ordonnanceurs sans ajustement sont incapables de produire des ordonnancements valides, contrairement à leurs homologues ajustés – au prix de durées d'exécution supérieures. Le réglage du facteur de bonus permet de tirer parti des deux aspects, et l'on s'aperçoit que, pour BF = 0,01, l'accélération peut atteindre 13 % avec sept emplacements, 12 % avec huit et 11 % avec neuf. Dans le pire des cas, la version ajustée est plus lente de 6 %, mais elle garantit la validité de l'ordonnancement produit. En somme, il est toujours possible de surpasser les heuristiques originales avec la technique d'ajustement. De plus, si l'on compare les figures 3.14 et 3.15, on remarque

qu'il n'y a pas de différence notable entre HEFT et SDC dans ce cas. Comme pour le TNR, les durées d'exécution et les accélérations décroissent avec la sévérité de la contrainte mémorielle ; en effet, les traitements de plusieurs macroblocs peuvent alors se recouvrir davantage. Inversement, avec une capacité de seulement quatre emplacements, la durée d'exécution est singulièrement élevée car la plupart des macroblocs doivent être traités séquentiellement.

Dans le deuxième jeu de simulations, les ordonnanceurs ont reçu un graphe de tâches décrivant le décodage de dix lignes de dix macroblocs. Les figures 3.16 et 3.17 illustrent les résultats. L'issue est similaire, si ce n'est que les heuristiques originales sont dans l'incapacité de générer des ordonnancements légaux avec une capacité mémorielle inférieure à dix-neuf emplacements, tandis que les variantes ajustées y sont aptes dès quatre emplacements.

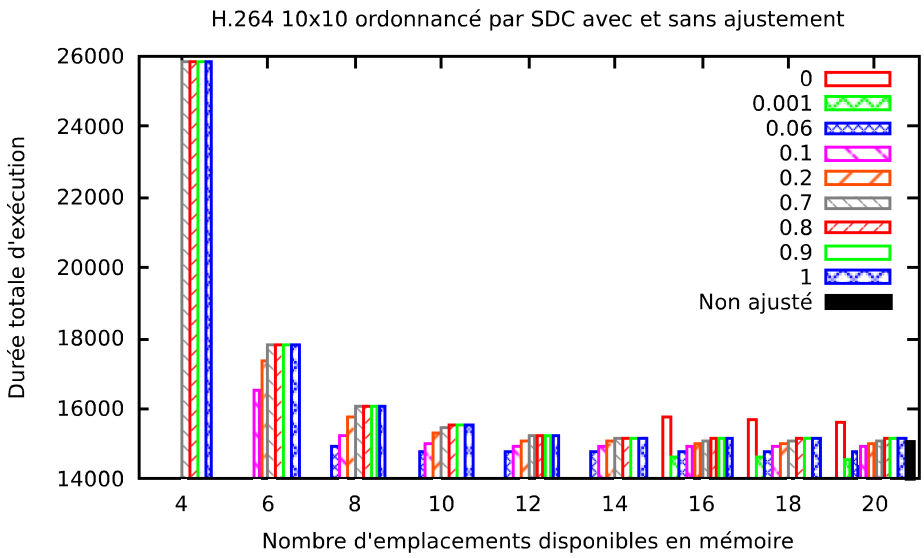


FIGURE 3.16 – *H.264 avec 10×10 macroblocs ordonnancé par SDC.*

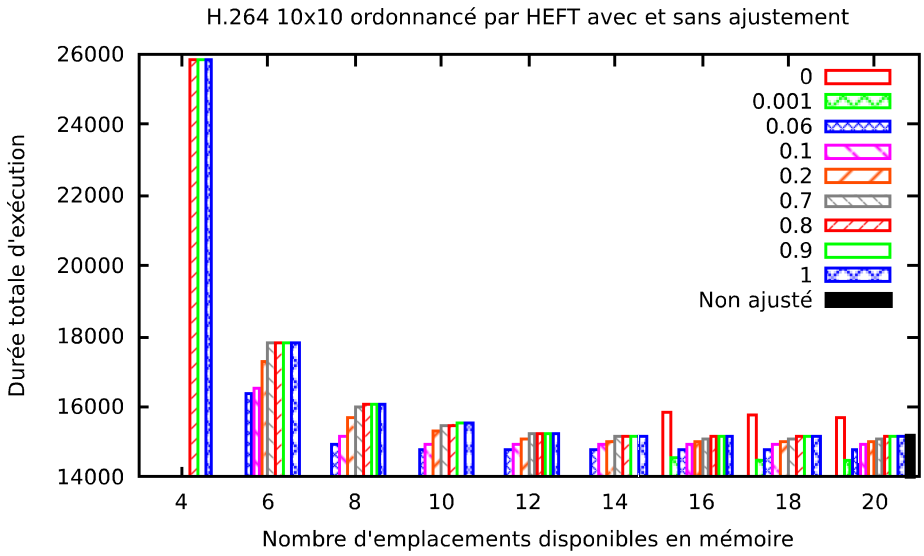


FIGURE 3.17 – *H.264 avec 10×10 macroblocs ordonnancé par HEFT.*

Le tableau 5 indique, pour chaque valeur du facteur de bonus, le nombre total d'ordonnancements qui ont nécessité un forçage de priorités, sur 10 000 tirages. Les résultats pour les valeurs de BF les plus élevées (0 et 2) confirment la qualité globale de l'ajustement effectué avant même que le forçage entre en jeu. La mesure lorsque BF tend vers 0 ne prend en compte que les cas où la capacité mémorielle est suffisante pour ne pas aboutir à une impasse.

Facteur de bonus	0	0,001	0,06	0,1	0,2	0,7	0,8	0,9	1
Nombre d'ordonnancements avec forçage	5454	5476	3048	1491	508	702	996	0	2

TABLEAU 5 – *Recours au forçage en fonction du facteur de bonus.*

L'ensemble des résultats soulignent la très grande proximité des performances de HEFT et SDC, ce qui démontre la capacité des solutions proposées dans ce chapitre à être appliquées à différentes heuristiques existantes de façon tout aussi bénéfique.

3.6 Conclusion

Ce chapitre a présenté des extensions aux algorithmes d'ordonnancement de liste classiques, tels que HEFT et SDC, pour prendre en compte les contraintes de mémoire. Cela repose sur un nouveau modèle comprenant des tâches mémorielles et impliquant un ajustement des priorités initiales. Le mécanisme d'insertion de tâches dans des créneaux d'inactivité a aussi été étendu à ce cas. Les expériences sur une application simple (TNR) ont montré qu'il est possible d'atteindre des accélérations d'au moins 20 %. Sur une application plus complexe (décodeur H.264), des cas de blocage des heuristiques standard ont été mis en évidence quand les contraintes mémorielles sont importantes. Seul un ajustement rigoureux des priorités permet d'éviter de telles impasses. En outre, le compromis entre durée totale d'exécution et consommation mémorielle a été exploré, et il ressort que la méthodologie proposée permet de trouver des ordonnancements qui surpassent les heuristiques originales sur ces deux critères.

Chapitre 4. Implémentation et aspects expérimentaux

Ce chapitre intermédiaire a pour objet de donner les bases nécessaires à la compréhension des conditions dans lesquelles ont été menés les développements et expérimentations dans le cadre de cette thèse. Dans un premier temps, il détaille les techniques couramment employées dans la conception des architectures ciblées, l'accent étant mis sur l'aspect logiciel, et les outils existants. Sont ensuite relatées les modalités pratiques de simulation, dans les plateformes embarquées en général, puis dans le cas particulier de STHORM. Enfin, les applications sur lesquelles sont basées les expériences menées dans les chapitres suivants sont présentées ici.

4.1 Co-conception matériel-logiciel

Lorsque l'on dispose d'une application (p. ex. H.264) décrite dans un langage de programmation séquentiel (p. ex. le langage C) et d'un modèle d'architecture (p. ex. STHORM), l'essentiel du travail consiste à dimensionner correctement la plateforme et à représenter le programme de manière à y faciliter son exécution [58]. Dans le cas où la représentation choisie est un modèle à flot de données, la première étape est le partitionnement de l'application : l'algorithme de calcul est décomposé en unités fonctionnelles (acteurs) cohérentes et équilibrées, soit simplement à partir de la connaissance qu'en a le concepteur, soit en s'aidant des informations de profilage recueillies lors de l'exécution du code séquentiel original. Cette opération, parfois longue, ne doit pas être négligée car elle détermine, entre autres, la granularité des acteurs dont l'importance est cruciale pour l'implémentation logicielle afin d'éviter les changements de contexte intempestifs. En effet, un découpage à grains trop gros risque de produire des invocations longues et d'aboutir à une monopolisation chronique des ressources ; au contraire, une granularité trop fine pourrait multiplier inutilement les tâches de contrôle et accroitre de façon déraisonnable la pression sur l'ordonnanceur.

Le dimensionnement de la plateforme peut comprendre, selon les cas, un nombre élevé de paramètres, mais dont les principaux se résument le plus souvent au nombre et aux types de processeurs d'une part, à la capacité des files inter-acteurs d'autre part. Pour ce qui est des unités de calcul, à la fois matérielles et logicielles, il s'agit d'assurer un compromis entre la surface de silicium, l'efficacité et, parfois, le risque de blocage. Pour les files, le compromis à atteindre se situe entre l'empreinte en mémoire, l'absorption de la latence – si deux acteurs successifs ont des cadences variables, augmenter la taille du tampon intermédiaire permet de masquer la gigue – et, encore une fois, le risque de blocage en cas de sous-dimensionnement. L'alternance de ces deux étapes, qui peuvent avoir lieu en parallèle, avec des phases de simulation forment le socle de la plupart des méthodologies de prototypage rapide.

4.1.1 Outils

Cette section illustre le paragraphe précédent en décrivant quelques cadriciels voués à la synthèse de programmes décrits à l'aide de différentes variantes de modèles à flot de données et ciblant les architectures visées.

4.1.1.1 PEDF

Le chapitre 2 ayant déjà abordé le modèle de programmation, il s'agit ici uniquement de présenter la chaine d'outils qui l'implémente: compilateur, logiciel système, etc. La figure 4.1 offre une vue d'ensemble de la pile logicielle de PEDF, hors compilateur et outils externes.

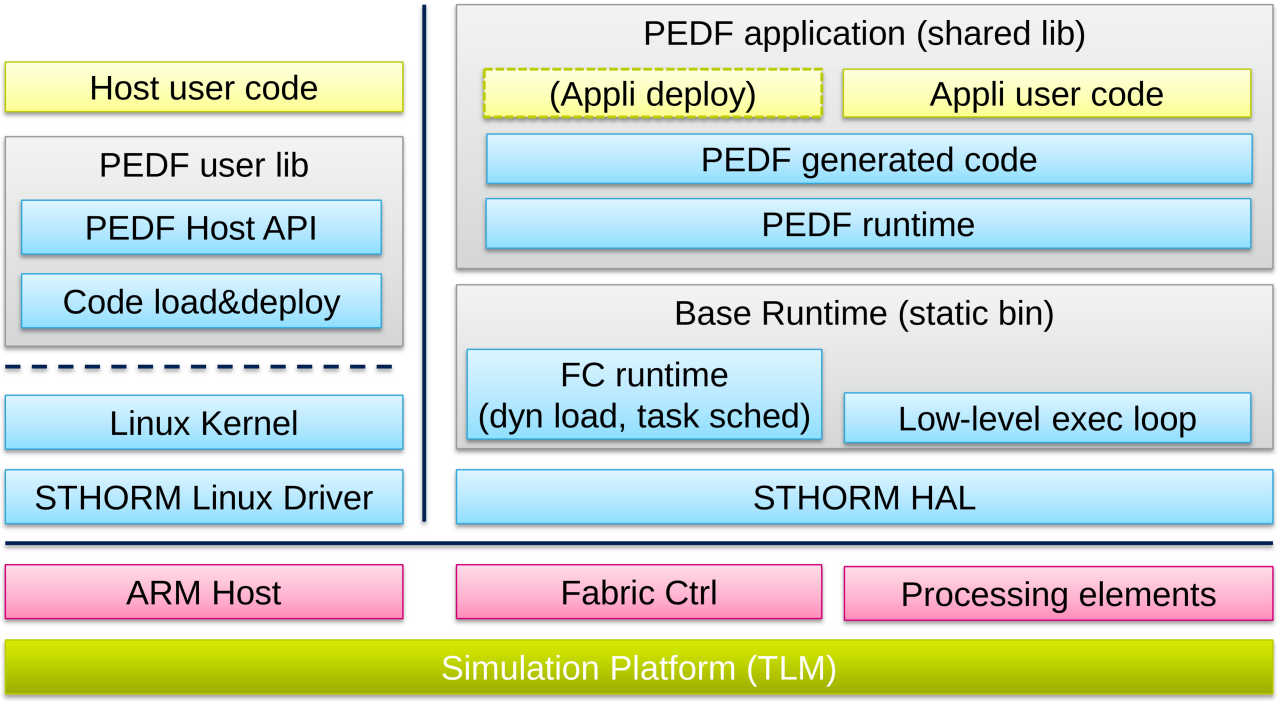


FIGURE 4.1 – *Pile logicielle de PEDF. La partie inférieure (boites roses et verte) représente le matériel simulé, et la partie supérieure le logiciel. La partie gauche correspond au processeur hôte, et la droite à la fabrique. Source: interne ST.*

Une application PEDF est une bibliothèque partagée composée du code de l'utilisateur ainsi que d'une couche sous-jacente générée par les outils, qui permet d'assurer l'intégration avec le logiciel système PEDF (*PEDF runtime*), lui aussi sous forme de bibliothèque partagée. Celui-ci expose à l'application des fonctionnalités de relativement haut niveau: allocation dynamique de la mémoire à tous les niveaux de hiérarchie, diagnostic pour le débogage, contrôle de l'exécution, configuration des compteurs de temps,...; il inclut également un moteur coopératif pour l'ordonnancement des fils d'exécution sur les PE d'ENCore.

Le logiciel système de base (*base runtime*) est quant à lui compilé en un binaire statique, chargé par le processeur hôte du SoC, qui offre des primitives de bas niveau pour la gestion des ressources matérielles de la fabrique: évènements, compteurs atomiques, mémoire, processeurs, etc.; il met également à disposition des fonctionnalités de plus haut niveau: ordonnancement de tâches sur le contrôleur de la fabrique, sémaphores et verrous. L'essentiel du code réside du côté du contrôleur de la fabrique (FC), ce qui signifie que les PE d'ENCore ne peuvent y faire appel directement mais doivent envoyer un message pour formuler leur requête. Enfin, il est en charge

de l'amorçage de la fabrique, de l'édition des liens et du chargement dynamiques du logiciel système PEDF et de l'application. Une fois les PE d'ENCore amorcés, ils tournent une boucle en attente d'un message indiquant la fonction à exécuter.

Le processeur hôte est supposé être un cœur ARM avec un noyau Linux. Dans le cas présent, l'hôte est modélisé par un simple processus POSIX simulant le chargement et le déploiement de l'application à l'aide d'une bibliothèque *ad hoc* sur laquelle le code de l'utilisateur vient se brancher.

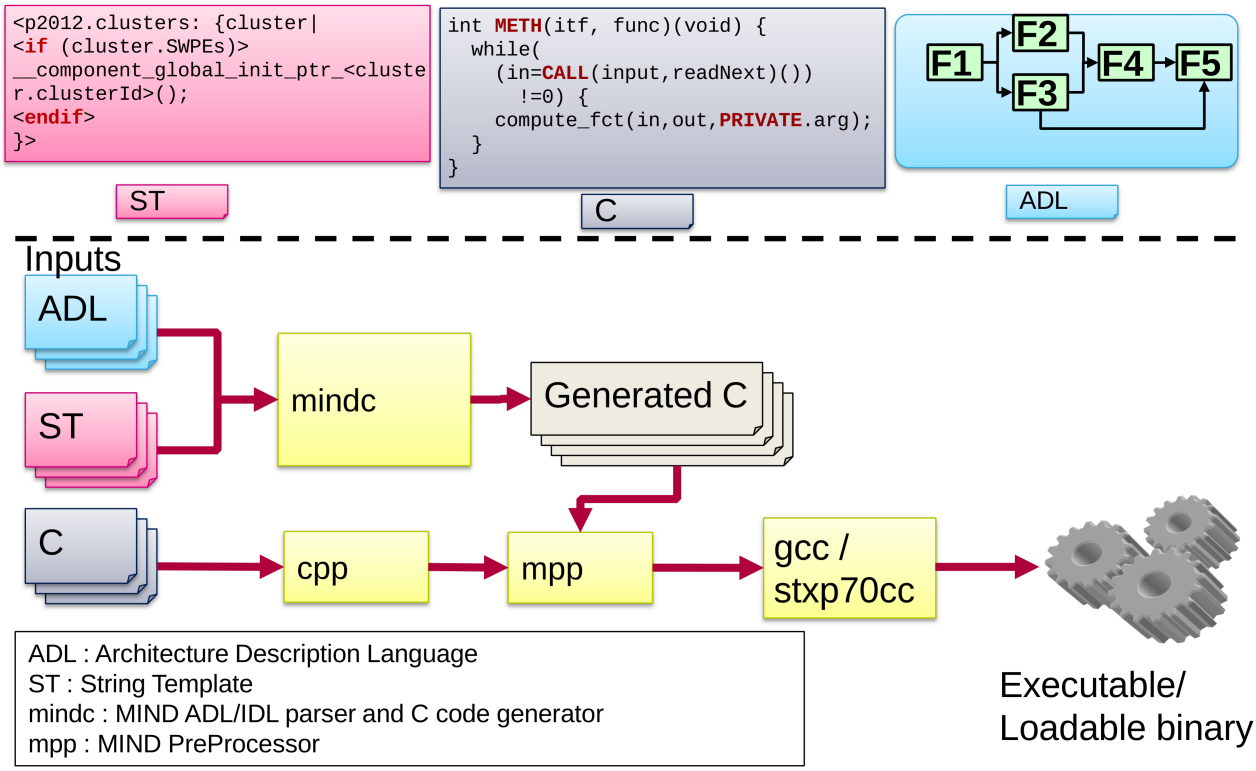


FIGURE 4.2 – *Chaine de compilation de PEDF. Source: interne ST.*

Comme le montre la figure 4.2, les bibliothèques et binaires décrits ci-dessus sont produits par un compilateur fondé sur le cadriciel de programmation orientée composants MIND[22], qui prend en entrée une description architecturale du graphe (*architecture description language*, ADL) et le code des acteurs (en langage C), pour générer l'application finale. La procédure de compilation, complexe et opaque, comprend plusieurs phases: les fichiers ADL sont utilisés par l'extension PEDF du compilateur MIND pour générer du code C à partir de patrons StringTemplate[23], tandis que le code des acteurs subit une première passe de préprocesseur C; l'ensemble est ensuite envoyé au préprocesseur MIND, avant de passer au compilateur C adéquat (cf. section 4.2.1), ce qui inclut une nouvelle passe de préprocesseur C. Par ailleurs, au cours du processus, les décisions de mappage statique sont prises de façon automatique – en l'absence d'intervention de l'utilisateur – ou semi-automatique via un script dans le langage Groovy[24] qui permet de déterminer notamment: l'assignation des contrôleurs aux SWPE et des filtres aux HWPE, la taille par défaut des files inter-acteurs et celles des piles, et l'assignation des attributs aux banques de registres.

22 http://mind.ow2.org/
23 http://www.stringtemplate.org/
24 http://groovy.codehaus.org/

Outre un système de construction peu maniable – nécessitant de nombreux fichiers de configuration éparpillés dans plusieurs modules interdépendants –, basé sur Gradle[25] et Maven[26], et une documentation lacunaire, PEDF souffre d'un certain nombre de limites et défauts. Parmi elles, on peut citer l'absence d'ordonnanceur pour les filtres et l'impossibilité d'assigner les contrôleurs aux SWPE dynamiquement. De plus, le support des filtres logiciels, bien qu'initialement prévu, n'a jamais été implémenté. Le chapitre 5 montre comment les travaux menés au cours de cette thèse ont permis de pallier ces manques. Enfin, l'édition des liens et le chargement dynamiques ont de lourdes conséquences : le déploiement est très lent, les niveaux d'indirection supplémentaires induisent un surcoût tout au long de l'exécution, et l'implémentation simpliste du chargeur ne permet ni la résolution bidirectionnelle des symboles, ni le placement des objets dans des zones différentes de la mémoire.

4.1.1.2 Orcc

En parallèle de la spécification de la norme RVC mentionnée au chapitre 1, des travaux de recherche ont été menés dans le but de développer un compilateur capable de synthétiser une application complète à partir d'une description RVC-CAL. Orcc [59] se présente sous la forme d'un environnement de développement intégré basé sur Eclipse[27]. Il prend en entrée un modèle de décodeur issu soit de la bibliothèque MPEG standard, soit d'une implémentation propriétaire optimisée. Il dispose de nombreux générateurs de code, à la fois pour des cibles matérielles et logicielles : C, Java, LLVM, Verilog, etc.

Il est à noter que, contrairement à PEDF, Orcc repose sur un modèle à flot de données défini formellement : RVC-CAL [60]. Dans ce modèle dérivé de DPN, chaque invocation correspond à une action sélectionnée par un ordonnanceur interne à chaque acteur en fonction de son état propre et d'un ensemble de règles d'invocation. Lorsque plusieurs actions sont activées en même temps, des priorités peuvent permettre de les départager pour savoir laquelle sera invoquée en premier. L'ensemble des contraintes d'invocation peut être synthétisé par un automate fini.

4.1.1.3 MAPS

De manière similaire, MPSoC Application Programming Studio [61] est un environnement de développement complet intégrant son propre modèle programmation, un générateur de code C, ainsi que des outils de trace et d'analyse de performance. En revanche, contrairement à Orcc, il cible explicitement les plateformes embarquées. L'application est décrite dans une extension du langage C qui permet de modéliser des processus KPN, des acteur SDF (cf. chapitre 2) et des files de communication. La génération de code se faisant exclusivement en C, seuls les processeurs programmables peuvent être ciblés.

4.1.1.4 Discussion

Dans le cadre de cette thèse, la cible matérielle privilégiée étant STHORM, l'essentiel des travaux ont été menés sur PEDF. Néanmoins, Orcc faisant figure de référence dans la recherche sur le décodage vidéo *via* les modèles à flot de données, quelques développements ont été effectués avec cet outil à des fins d'évaluation. Un décodeur vidéo simple a été synthétisé et une implémentation

25 http://www.gradle.org/
26 http://maven.apache.org/
27 https://www.eclipse.org/ide/

monoprocesseur a été portée sur un ISS autonome (hors de la plateforme de simulation de STHORM) avec succès. Cependant, les générateurs de code présents n'étant pas adaptés pour les cibles embarquées multiprocesseurs, il a été conclu que l'effort nécessaire à la synthèse d'une application fonctionnelle pour STHORM – ce qui aurait, à priori, impliqué la génération de code PEDF – surpassait les bénéfices industriels potentiels.

4.2 Simulation

Cette section relate quelques généralités sur la modélisation et la simulation des architectures embarquées pour SoC, puis s'attarde sur le cas particulier de STHORM.

Différents niveaux d'abstraction sont généralement proposés selon le compromis souhaité entre précision et vitesse de simulation. En effet, les modèles de plus haut niveau sont plus efficaces pour des exécutions rapides mais ne permettent pas de prendre en compte toutes les spécificités du matériel; au contraire, ceux de plus bas niveau sont plus adaptés pour l'introspection et l'exploration architecturales mais souffrent de lenteurs parfois rédhibitoires pour des applications complexes.

Les outils les plus couramment utilisés pour la simulation des circuits [62] sont l'environnement SystemC[28] et la bibliothèque de modélisation au niveau transactionnel (*transaction-level modeling*, TLM) qui en est une surcouche très usitée permettant la description des interactions au sein du système simulé tout en masquant les détails d'implémentations internes à chaque module. Ces briques logicielles sont bien adaptées pour la modélisation des systèmes à haut niveau de concurrence, ce qui est le cas des circuits électroniques où tous les blocs matériels fonctionnent en parallèle. SystemC offre un moteur de processus et de fils d'exécution en espace utilisateur, qui permet à chaque module de s'exécuter indépendamment et d'être ordonnancé lorsqu'il reçoit un signal. En revanche, le simulateur lui-même s'exécute de façon séquentielle sur la machine hôte, ce qui empêche de tirer parti du parallélisme et constitue un frein important pour l'augmentation des vitesses de simulation [63].

Enfin, les unités de calcul programmables sont la plupart du temps modélisées selon deux niveaux de précision: à l'instruction ou au cycle près. Les simulateurs de jeux d'instructions (*instruction-set simulators*, ISS) appartiennent à la première catégorie et permettent l'exécution de code natif – c'est-à-dire compilé pour la cible réelle – sans s'encombrer des détails de la micro-architecture (pipeline, accès aux caches, etc.).

Une plateforme TLM peut inclure (ou non) un modèle de performance et, le cas échéant, sa précision peut varier fortement selon les besoins. Si une évaluation de performance poussée est requise, alors la précision peut descendre jusqu'au cycle. Un ISS permet au moins une précision à l'instruction près et a souvent la possibilité d'estimer le nombre de cycles correspondants.

4.2.1 STHORM

La plateforme de STMicroelectronics ciblée par les expériences présentées au chapitre 6 dispose de plusieurs simulateurs à différents niveaux d'abstraction, fondés notamment sur les technologies présentées aux paragraphes précédents, et permettant, en théorie, de faciliter le cycle de développement logiciel à chaque étape. L'avantage réside notamment dans le fait de pouvoir cibler

28 http://www.accellera.org/downloads/standards/systemc

différents simulateurs sans avoir à modifier le code de l'application. Les paragraphes suivants présentent successivement les différents niveaux de modélisation proposés, du plus haut au plus bas.

Un premier simulateur est dédié à l'exploration applicative (*application exploration*, Apex). Il s'agit de simulation fonctionnelle du modèle de programmation de PEDF, et non du matériel. Le code généré est un binaire x86 et les SWPE sont des fils d'exécution SystemC. Le niveau d'abstraction étant donc très élevé, il n'y a aucune notion de temps. Ce simulateur s'utilise au tout début du processus de développement, à des fins de prototypage et de débogage préliminaire de l'application.

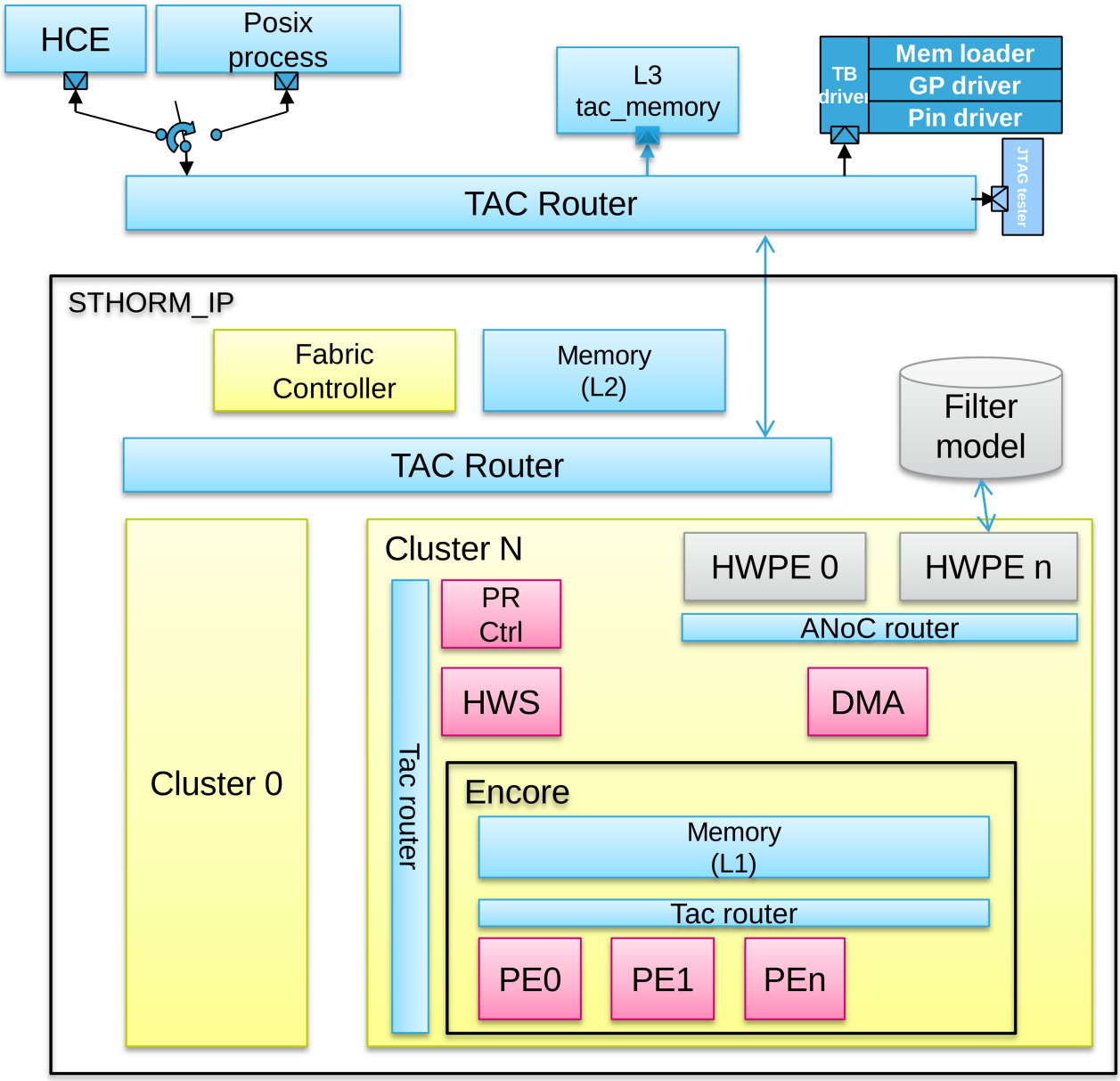


FIGURE 4.3 – *Vue d'ensemble du simulateur TLM de STHORM. Source: interne ST.*

STHORM dispose ensuite d'une plateforme de simulation TLM, représentée par la figure 4.3, qui modélise le matériel à un niveau d'abstraction relativement élevé et permet une évaluation de performances approximative. Le processeur hôte du SoC, dont le fonctionnement matériel a ici peu d'importance, est simulé à très haut niveau par le biais d'un simple processus POSIX séparé

(en haut à gauche sur la figure), qui communique avec le processus SystemC *via* des tubes. Les SWPE (représentés en rose et notés simplement PE sur la figure) sont cette fois-ci modélisés par des ISS qui exécutent donc du code natif STxP70 obtenu par compilation croisée. La partie utilisateur des HWPE (en gris) – c'est-à-dire les filtres – bénéficie d'une modélisation fonctionnelle à haut niveau (C++ x86) qui permet un prototypage rapide ne nécessitant pas la synthèse du matériel. Tous les autres modules (DMA, HWS, etc.) sont modélisés au niveau registre et disposent d'annotations temporelles dont la précision est paramétrable, ce qui permet par exemple d'estimer les temps d'accès à la mémoire, mais sans tenir compte de la contention.

Ce simulateur offre un compromis acceptable entre vitesse d'exécution et précision du modèle. Par exemple, pour un temps d'exécution réel estimé à 324 s, la durée de simulation est de 371 000 s, soit un rapport de l'ordre de 1000. Le modèle de performance prend essentiellement en compte le temps de calcul dans les ISS, les accès aux HWPE et la latence introduite par les interconnexions, cette dernière fonctionnalité étant débrayable. Ce sera donc cette plateforme TLM qui sera retenue pour les expériences à venir.

Enfin, il existe deux autres environnements de simulation qui n'ont pas été utilisés dans le cadre de ces travaux, mais qui sont mentionnés ici par souci d'exhaustivité. La co-simulation entre modèle de description matérielle de bas niveau (*register-transfer level*, RTL) et TLM permet une analyse de performances précise et est utilisée notamment pour la vérification. L'émulation matérielle sur FPGA permet une vérification plus rapide et une analyse des performances réelles après synthèse.

4.2.2 Durée des filtres matériels

Comme l'a mentionné la section précédente, la partie définie par l'utilisateur des HWPE est simulée de façon fonctionnelle à un haut niveau d'abstraction, ce qui implique en particulier qu'elle ne dispose d'aucun modèle de temps. En pratique, seules les opérations gérées par l'enveloppe des HWPE (accès aux files matérielles, accès aux banques d'attributs, commande d'invocation, etc.) sont modélisées. Il apparaît donc nécessaire, dans le cadre des travaux menés au cours de cette thèse, d'introduire un support pour la modélisation temporelle au niveau des filtres matériels.

Pour ce faire, l'approche retenue a été celle d'une interface simple permettant à chaque filtre de fournir une durée de référence, supposée être une valeur moyenne, qui sert de base au calcul d'un délai ajouté au temps simulé. Ce délai comprend un facteur aléatoire issu d'un tirage selon une loi gaussienne ou bêta, ou émanant directement de l'appel d'une fonction non linéaire (générateur pseudo-aléatoire), et un facteur constant. On note :

- rct : le temps de calcul de référence fourni par le filtre ;
- cdf : la composante constante du facteur de délai fournie par l'application et appliquée à tous les filtres ;
- rdf : la composante aléatoire du facteur de délai dont la source est sélectionnée au niveau de l'application et est valable pour tous les filtres ;
- df : le facteur de délai complet ;
- td : le délai total.

La composante constante modélise l'accélération moyenne introduite par l'implémentation matérielle par rapport au logiciel de référence. La composante aléatoire est là pour rendre compte de la variabilité et de l'imprédictibilité. L'interface prend la forme d'une fonction qui peut être appelée depuis un point quelconque du code du filtre, éventuellement plusieurs fois par invocation. Chaque appel produit un délai différent obtenu de la manière suivante :

$$\mathrm{df} = \mathrm{cdf} \times \mathrm{rdf}$$
$$\mathrm{td} = \mathrm{df} \times \mathrm{rct}$$

Il est à noter que, selon les valeurs choisies, cette modélisation peut avoir un impact sensible sur les temps de simulation.

4.3 Applications

Cette section décrit plus en détails les deux applications à flux de données utilisées pour les expérimentations : TNR et H.264.

4.3.1 TNR

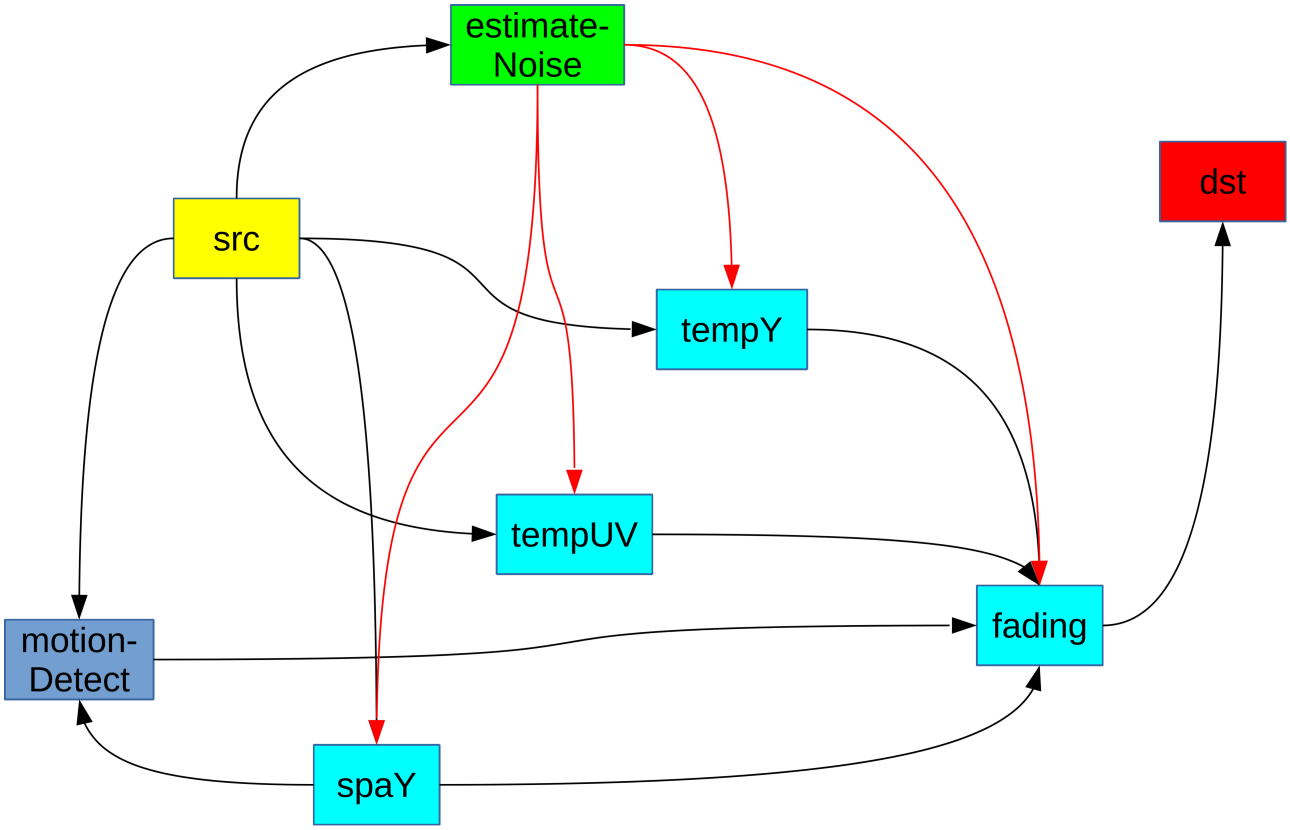


FIGURE 4.4 – *Graphe de l'application TNR. Les flèches noires représentent les canaux de données, et les rouges ceux de paramètres.*

Développée par STMicroelectronics, l'application de réduction de bruit temporel (*temporal noise reduction*, TNR) est un algorithme d'amélioration de la qualité d'image statique, au sens où les différentes invocations d'un même acteur font sensiblement le même calcul et les durées d'exécution ne varient donc pas, et peu complexe – la structure du graphe et les calculs sont simples. Il prend en entrée une séquence vidéo et traite chaque image ligne par ligne de pixels. Les lignes

étant indépendantes du point de vue de l'algorithme, un haut niveau de parallélisme potentiel est disponible.

La figure 4.4 représente le graphe simplifié du TNR. Il est composé de huit acteurs dont cinq sont affectés au calcul, un aux reconfigurations et deux aux transferts DMA. `spaY`, `tempUV`, `tempY` et `fading` procèdent au traitement des lignes sous le contrôle d'un paramètre d'estimation du niveau de bruit produit par `estimateNoise` à partir de l'analyse de l'image précédente; le traitement appliqué par `motionDetect` est quant à lui indépendant des données. `src` et `dst` sont chargés respectivement du transfert des données entrantes depuis la mémoire externe vers la mémoire locale, et vice-versa. La reconfiguration effectuée par `estimateNoise` a lieu au point de repos entre deux images; le paramètre qu'il produit est de type fonctionnel et n'est pas indicatif (cf. chapitre 2).

L'implémentation sur STHORM fait uniquement appel à des filtres matériels. L'hôte est chargé de la lecture des fichiers de configuration indiquant les caractéristiques de la séquence vidéo à traiter, de l'allocation en mémoire dans un espace d'adresse accessible par la fabrique pour y échanger les paramètres, ainsi que du déploiement et du contrôle de l'application sur la fabrique. Cette dernière reçoit pour chaque image une commande de l'hôte qui déclenche l'exécution du graphe, après reconfiguration de celui-ci.

Le portage du TNR en PEDF a nécessité un certain nombre d'opérations. En premier lieu, il était nécessaire de fournir la description des filtres (ports, attributs et état) et des modules (contrôleur, filtres et interconnexions) dans le langage ADL de MIND étendu pour PEDF. Le code des contrôleurs devait être écrit en C enrichi de macros MIND et PEDF. Des appels à ces primitives devaient également être insérés dans le code des filtres, de manière à gérer la lecture et l'écriture des attributs, le transfert des données, ainsi que l'état (les données privées) de chaque filtre. La description initialement monolithique de l'application a dû être scindée en trois modules: un pour les calculs et un pour chaque transfert DMA. Côté fabrique, la communication des paramètres avec l'hôte ainsi que l'interprétation des commandes a été implémentée. De même, côté hôte, l'initialisation et le déploiement de l'application, ainsi que la communication avec la fabrique.

Enfin, une dernière étape de modularisation a été nécessaire pour faire correspondre la description pour PEDF au modèle d'exécution. En effet, comme l'a indiqué le chapitre 3, celui-ci impose qu'un module ne comprenne qu'un seul filtre de façon à se conformer à la définition d'un acteur. Chacun des six filtres de calcul a ainsi été placé dans son propre module avec, par conséquent, son propre contrôleur. Dès lors, il devenait possible de contrôler chaque filtre indépendamment.

4.3.2 H.264

Dans la vaste classe des applications à flux de données, on s'intéresse plus précisément à celles présentant un fort dynamisme du fait de leur variabilité et de leur imprédictibilité, notamment en termes de reconfigurations et des durées d'exécution qui en résultent. Les codecs vidéo étant bien représentatifs de cette catégorie, la deuxième application retenue pour les expérimentations est un décodeur H.264. Les travaux ont porté sur une implémentation propriétaire de STMicroelectronics déjà adaptée pour PEDF: les opérations de portage requises par le TNR n'ont donc pas été nécessaires cette fois-ci.

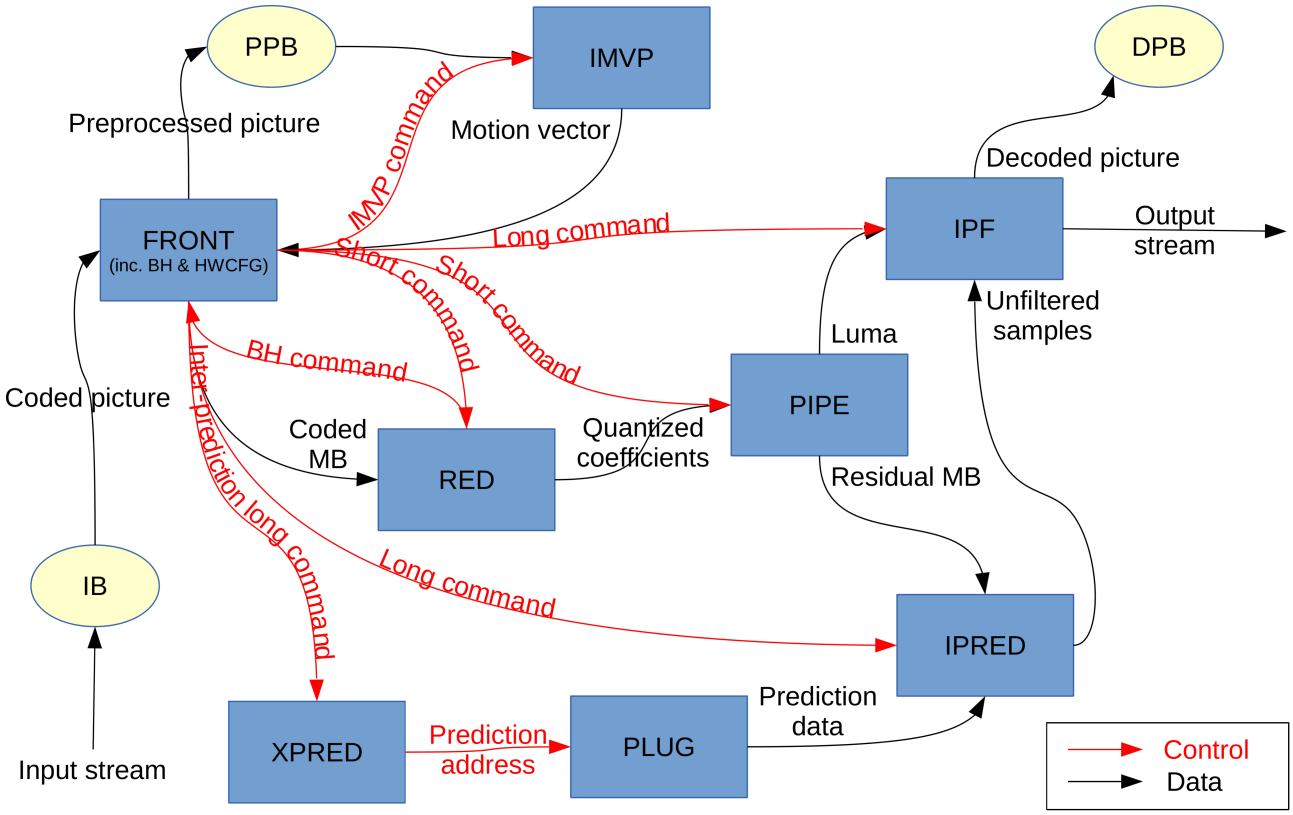


FIGURE 4.5 – *Graphe du décodeur H.264. Les acteurs sont représentés par des boites bleues, les mémoires tampons par des ellipses jaunes. Les flèches noires représentent les canaux de données, et les rouges ceux de paramètres.*

La figure 4.5 représente le graphe du décodeur H.264 utilisé. Il est composé de huit acteurs, dont un macro-acteur – c'est-à-dire un acteur ne respectant pas strictement le modèle d'exécution, pour des raisons de simplification d'implémentation. Le décodage d'une image commence par l'hôte qui écrit les données d'entrée dans le tampon intermédiaire (*intermediate buffer*, IB). FRONT demande alors à BH de lire les en-têtes de l'image et de la tranche, puis à HWCFG d'envoyer les commandes afférentes (`picture_cmd` et `slice_cmd`) aux autres acteurs afin de les reconfigurer. Ensuite, FRONT demande à BH de lire l'en-tête du prochain macrobloc à décoder pour savoir s'il est codé en inter- ou en intra-prédiction. Dans le premier cas, FRONT envoie d'abord à IMVP la commande pour calculer le vecteur mouvement, puis demande à HWCFG d'envoyer les commandes `short_cmd` et `inter_long_cmd` respectivement à RED et PIPE d'une part, et à XPRED, IPRED et IPF. Dans le second cas, FRONT demande immédiatement à HWCFG d'envoyer les commandes `short_cmd` et `intra_long_cmd`, cette dernière allant uniquement à IPRED et IPF.

Comme pour le TNR, des travaux de modularisation ont dû être menés pour faire correspondre la description en PEDF avec le modèle d'exécution. Comme le montre la figure 4.5, chacun des filtres a été encapsulé dans son propre module et doté de son propre contrôleur, à l'exception de BH et HWCFG qui ont été regroupés dans le macro-acteur FRONT pour faciliter l'implémentation. Par ailleurs, la description d'origine prévoyait une granularité d'invocation au niveau image, ce qui est bien adapté pour un mappage totalement matériel mais fort peu approprié pour les filtres implémentés en logiciel. En effet, quand le nombre d'unités de calcul est inférieur au nombre d'acteurs, il est important de garder des invocations courtes de manière à assurer une avancée homogène de

l'exécution de l'ensemble du graphe et à limiter la taille des files inter-acteurs; autrement, de longues invocations risqueraient de monopoliser les ressources et d'augmenter le risque d'impasse. Un raffinement de cette granularité au niveau macrobloc s'est donc imposée pour les acteurs candidats à l'exécution logicielle, à savoir IPRED et IPF. Du fait de l'effort de développement important exigé par le passage du matériel au logiciel (cf. chapitre 5), il n'était en effet pas envisageable de l'effectuer pour la totalité des filtres; un choix a donc dû être fait. IPRED a été retenu pour sa variabilité élevée, largement représentative des applications de ce type: parmi les quatre commandes qu'elle reçoit (`picture_cmd`, `slice_cmd`, `intra_long_cmd`, `inter_long_cmd`) les différences en termes de durée d'invocation atteignent plusieurs ordres de grandeur, comme le montre le tableau; de plus, la variabilité se manifeste y compris pour une même commande, où les temps d'exécution peuvent s'écarter de 10 % de leur valeur moyenne. IPF a été sélectionné pour des raisons similaires, ainsi que pour les accès DMA qu'il effectue.

Commande	Durée moyenne en cycles
`picture_cmd`	<1000
`slice_cmd`	<1000
`intra_long_cmd`	50 000
`inter_long_cmd`	200 000

TABLEAU 6 – *Durée moyenne d'une invocation d'IPRED en fonction de la commande reçue.*

Enfin, FRONT, IPRED et IPF ont été dotés de paramètres au sens du modèle d'exécution. Initialement, pour tous les acteurs, les commandes étaient traitées comme des attributs PEDF qui étaient mis à jour, non pas à des points de repos précis, mais tout au long des invocations *via* le mécanisme de contrôle dynamique décrit au chapitre 3. Les quatre commandes précitées ont donc été modélisées sous forme de paramètres pour ces trois acteurs, de façon à être communiquées à chaque point de repos. Dans le cas de FRONT, celui-ci n'étant pas un acteur au sens strict du terme, une adaptation a dû être faite: les fonctions de travail de ses filtres et de son contrôleur étant conçues pour être appelées une seule fois par image, une solution de contournement était nécessaire pour permettre l'échange des paramètres selon les règles édictées par le modèle d'exécution. L'idée retenue a été de n'appeler effectivement le contrôleur et les filtres qu'une seule fois par image, tout en introduisant des pseudo-invocations à raison d'une par commande. Ainsi, vu de l'extérieur – en particulier de l'ordonnanceur – les invocations et le transfert des paramètres entre acteurs respectent bien le modèle d'exécution, alors même que le fonctionnement interne de FRONT n'a pas changé.

4.4 Conclusion

Ce chapitre a présenté l'environnement de travail dans lequel a évolué cette thèse: la méthodologie de développement, les outils de simulation et les applications utilisées. On notera que le choix des outils et des applications a été largement dicté par des impératifs liés au contexte industriel de la thèse. Ainsi, STHORM étant clairement la cible privilégiée de ces travaux, seules les ressources disponibles en interne à STMicroelectronics pouvaient être utilisées en pratique, restreignant l'ho-

rizon aux possibilités énoncées précédemment. Le recours à d'autres outils de développement et de simulation, ou à d'autres applications, aurait exigé un effort d'adaptation inabordable.

Fort de ces prémisses, le chapitre suivant sera en mesure de détailler les contributions apportées dans le domaine de la gestion et de l'ordonnancement sur des architectures hybrides d'applications dynamiques, telles que H.264, décrites dans des modèles à flot de données.

Chapitre 5. Contributions sur le logiciel système

Les deux chapitres précédents ayant décrit respectivement le modèle d'exécution proposé et la manière dont il a été employé vis-à-vis des applications visées et des infrastructures de développement de STHORM, celui-ci vient compléter la perspective des travaux menés au cours de la thèse en présentant ses contributions logicielles. La première section aborde l'aspect ordonnancement dynamique, complémentaire avec l'approche statique du chapitre 3. Les deux suivantes concernent l'implémentation du support des filtres logiciels dans PEDF et les questions afférentes. La quatrième soulève le problème du placement des acteurs, et enfin les deux dernières se concentrent sur l'intégration dans la pile logicielle existante des contributions présentées dans le reste du chapitre et l'adaptation des applications qu'elles impliquent.

5.1 Ordonnancement et gestion dynamique

Les applications à flux de données modernes telles que les décodeurs vidéo démontrent des niveaux élevés de variabilité, notamment en ce qui concerne les durées d'exécution, et nécessitent des reconfigurations. Par exemple, dans le cas du décodeur H.264 présenté au chapitre précédent, pour un même acteur, la durée de deux invocations successives peut varier de plusieurs ordres de grandeur. De nombreux travaux récents [64]–[66] ont mis en évidence que, dans ce contexte, la seule solution viable pour traiter ces contraintes en matière d'ordonnancement est le recours à une approche dynamique. En effet, il n'est en pratique généralement pas possible de caractériser totalement statiquement de telles applications en termes de temps d'exécution et de fréquence d'activation. De plus, la complexité des architectures multicœurs embarquées rend la plupart du temps l'ordonnancement dynamique préférable. Néanmoins, jusqu'à il y a peu, les recherches dans le domaine de l'ordonnancement multiprocesseur des modèles à flot de données étaient pour l'essentiel circonscrites au cas statique [67]–[69]. Certaines études [70], [71] ont ciblé spécifiquement les plateformes hétérogènes, mais elles ne présentent pas de techniques nouvelles visant à prendre en compte tout le dynamisme des applications. Dans le cadre de RVC, des méthodes d'ordonnancement pour les modèles DPN ont été proposées [72]–[75] mais l'assignation des acteurs aux ressources de calcul n'est jamais totalement dynamique. La contribution présentée dans cette section vient combler ces manques.

Le but est d'ordonnancer efficacement des applications décrites sous forme de graphes à flot de données sur des plateformes embarquées multicœurs hétérogènes telles que STHORM, sous les hypothèses suivantes :

– Les implémentations des filtres sont mutuellement exclusives, c'est-à-dire qu'un filtre donné peut être exécuté soit en matériel soit en logiciel, mais pas les deux (cf. section 5.4).

- L'ensemble des HWPE disponibles sur la plateforme cible et celui des filtres exécutables matériellement sont en bijection. En d'autres termes, chaque filtre matériel est assigné statiquement à un HWPE, et chaque HWPE ne peut exécuter qu'un seul filtre.
- Tous les contrôleurs sont implémentés en logiciel pour des raisons de flexibilité et de facilité d'intégration avec le logiciel système et le code de l'utilisateur.
- Tous les contrôleurs peuvent être exécutés sur tous les SWPE, ces derniers étant symétriques.
- Les changements de contexte étant coûteux et rendus facultatifs par le modèle d'exécution, ils ne sont pas autorisés. De ce fait, chaque SWPE dispose d'un unique fil d'exécution pour à la fois l'ordonnanceur et les invocations d'acteurs.

L'ordonnanceur doit assurer l'arbitrage entre les acteurs partageant les mêmes ressources, garantir la bonne application du modèle d'exécution (règles d'activation, etc.) et éviter, dans la mesure du possible, les impasses. Il possède les caractéristiques et propriétés suivantes :

- distribution symétrique sur tous les SWPE afin d'assurer une plus grande réactivité, simplifier l'implémentation – le fait que la mémoire locale soit partagée étant un point crucial à cet égard – et garantir une certaine efficacité quant au passage à l'échelle ;
- coopération – ou, de façon équivalente, absence de préemption –, ce qui signifie que chaque tâche[29] doit rendre la main explicitement à l'ordonnanceur, ce dernier n'ayant pas la capacité d'interrompre un acteur en cours d'exécution ;
- agnosticisme vis-à-vis des transferts de données, ceux-ci se situant dans le domaine KPN, tandis que l'ordonnanceur évolue uniquement dans l'espace DPN et n'en a donc aucune connaissance.

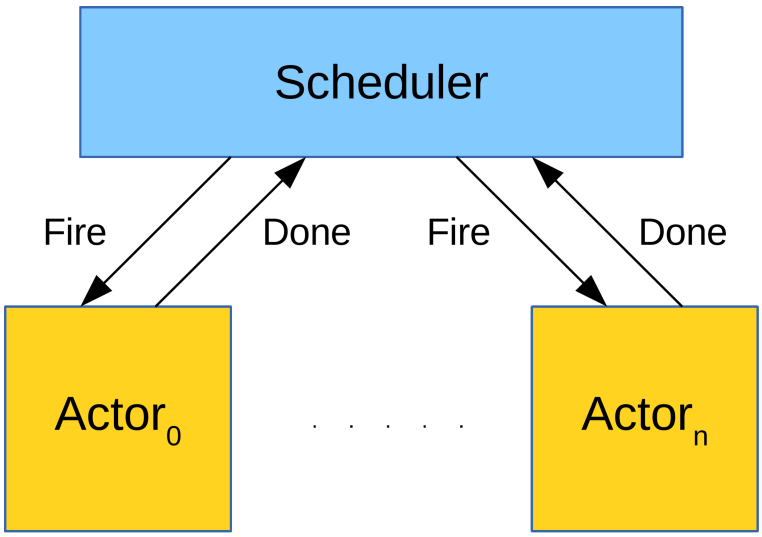


FIGURE 5.1 – *Relations entre ordonnanceur et acteurs.*

Puisque plusieurs acteurs peuvent s'exécuter de manière concurrente sur le même processeur, la principale attribution de l'ordonnanceur est de décider quel va être le prochain à être exécuté sur chacun d'eux. Bien qu'il ne soit pas responsable de leur transmission d'acteur à acteur, l'ordon-

29 Dans tout ce qui suit, le terme *tâche* désigne l'invocation d'un acteur.

nanceur a néanmoins la charge de s'assurer que les paramètres requis par l'un d'eux sont bien disponibles avant de l'invoquer. De plus, il est supposé capable d'interpréter les paramètres indicatifs décrits au chapitre 2 dans le but d'accroitre la pertinence de ses décisions.

La figure 5.1 illustre les échanges possibles entre l'ordonnanceur et les acteurs. Le premier surveille l'état des paramètres et, lorsqu'un acteur est prêt – c'est-à-dire que sa règle d'activation est satisfaite –, émet le signal `fire` en guise d'invocation. Les seconds, en fin d'invocation, envoient quant à eux le signal `done` pour signifier à l'ordonnanceur que les éventuels paramètres de sortie ont été produits et qu'ils peuvent potentiellement être à nouveau ordonnancés.

Une *étape* d'ordonnancement est définie comme une succession d'actions accomplies par l'ordonnanceur et comprenant les deux opérations suivantes dans cet ordre :

1. sélectionner les prochains acteurs à ordonnancer ;
2. invoquer un acteur.

Au cours d'une étape, l'ordonnanceur peut se trouver dans deux états distincts, comme le montre la figure 5.2 : inactif (*idle*) lorsqu'un acteur a été invoqué, actif lorsque la précédente tâche est achevée.

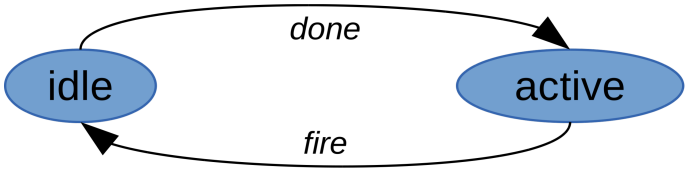


FIGURE 5.2 – *États de l'ordonnanceur.*

Chaque acteur dispose lui aussi de son propre état au niveau de l'ordonnanceur, qui comprend deux composantes. La première, représentée par la figure 5.3, détermine son statut vis-à-vis de l'ordonnanceur : désactivé – il ne peut pas être ordonnancé –, inactif – il pourra être ordonnancé dès qu'il sera prêt – ou actif – il est en cours d'exécution ou sur le point d'être invoqué. La deuxième indique s'il est prêt ou non, c'est-à-dire si ses paramètres d'entrée sont tous disponibles ou s'il en manque. Un acteur ne peut être ordonnancé que si son état est inactif et prêt.

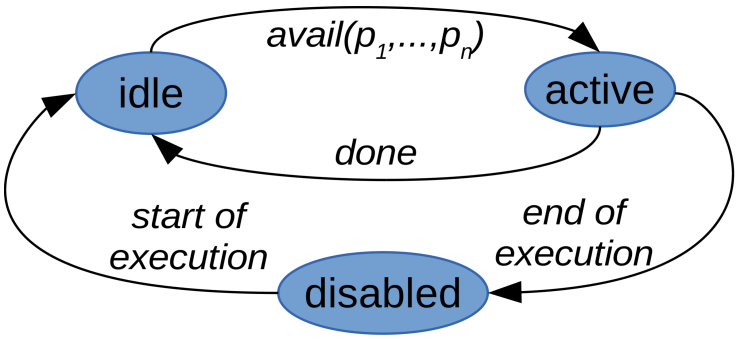


FIGURE 5.3 – *États d'un acteur du point de vue de l'ordonnanceur.*

Quand l'ordonnanceur reçoit un signal `done`, il met immédiatement l'acteur émetteur dans l'état inactif. Ensuite, dès que tous ses paramètres d'entrée (p_1,..., p_n) sont disponibles, il est déclaré prêt à être ordonnancé. Le moment venu, il est introduit dans une file de tâches globale permettant un équilibrage de charge entre tous les PE sur lesquels tourne l'ordonnanceur. Par défaut et en l'ab-

sence d'indications à cet égard fournies par l'application, tous les acteurs sont initialement inactifs. L'état désactivé n'est pour sa part atteint que par les acteurs qui ont terminé leur exécution, c'est-à-dire qui ont déjà effectué la totalité du travail requis par la commande émise par l'hôte. La transition ne peut avoir lieu qu'à la fin d'une invocation et est déclenchée par l'acteur lui-même étant donné que la nature de cette décision est essentiellement spécifique à chaque application. Cet état ne peut être quitté qu'au début de l'exécution d'une nouvelle commande.

Dans le modèle d'exécution, deux types de dépendances coexistent : celles liées aux paramètres et celles associées aux données. Les premières sont gérées par l'ordonnanceur et ne peuvent pas bloquer puisqu'elles sont régies par le modèle DPN ; en revanche, les secondes obéissent au modèle KPN et ne peuvent pas être prises en charge par l'ordonnanceur, sachant que les débits ne sont pas connus avant l'invocation. Les dépendances de données entre filtres matériels sont gérées *via* les mécanismes à flot de données supportés nativement par l'architecture ; lorsqu'au moins un filtre logiciel est impliqué, un support supplémentaire au niveau des outils est requis (cf. section 5.3). Les dépendances paramétriques sont conduites, au niveau de l'ordonnanceur, par le biais de compteurs, à raison d'un par paramètre, représentant le nombre de jetons disponibles dans chaque file. Ainsi, pour un acteur donné, déterminer s'il est prêt revient à s'assurer que les compteurs liés à ses paramètres d'entrée ont tous une valeur supérieure à 1.

L'algorithme 2 décrit plus avant le fonctionnement de l'ordonnanceur en détaillant la séquence d'opérations que constitue une étape d'ordonnancement. Il se décompose en trois phases :

1. comptabilisation des paramètres passés, vérification des règles d'activation et mise à jour des états ;
2. mise en file des acteurs prêts – cette phase, ici très simple, est d'une importance capitale car c'est elle qui accueillera les différentes politiques d'ordonnancement ;
3. invocation d'un acteur.

Cet algorithme se voulant générique, il fait volontairement l'impasse sur la stratégie d'ordonnancement adoptée : nombre et nature des files de tâches, ordre dans lequel les acteurs sont mis en file, etc. Les contraintes sur les politiques d'ordonnancement possibles seront décrites à la section 5.1.2 et une analyse comparée d'une sélection d'entre elles, adossée à des expérimentations, sera présentée au chapitre 6. En outre, la gestion des accès concurrents et du risque d'interblocage sont pris en compte à chaque phase. Les files de paramètres – comme celles de données – ne nécessitent pas de verrous car elles n'ont chacune qu'un seul producteur et un seul consommateur [76] et ceux-ci utilisent une attente active. En revanche, les files de tâches étant, dans le cas général, à la fois lues et écrites par tous les PE du fait de la distribution symétrique de l'ordonnanceur, elles nécessitent chacune un sémaphore binaire pour garantir l'exclusion mutuelle. La partie de l'état d'un acteur indiquant s'il est prêt doit pouvoir être lue et écrite atomiquement pour la phase 2. De même, l'acquittement des signaux `done` doit être atomique. Ces deux derniers points dépendant largement de l'architecture, ils seront développés à la sous-section suivante.

```
Début étape d'ordonnancement

# Phase 1
Pour chaque signal done en attente :
     Acquitter le signal
     Mettre à jour les paramètres disponibles
     Mettre l'acteur émetteur dans l'état inactif, s'il n'est pas
désactivé
     Si pas de paramètres d'entrée ou tous disponibles :
          Déclarer prêt
     Fin Si
     Pour chaque paramètre produit :
          Pour chaque file associée :
               Si compteur = 1 :
                    Marquer le paramètre disponible
                    Si tous les paramètres sont disponibles et
l'acteur est inactif :
                         Le déclarer prêt
                    Fin Si
               Fin Si
          Fin Pour
     Fin Pour
Fin Pour

# Phase 2
Pour chaque acteur prêt :
     Mettre dans la file de tâches ad hoc
Fin Pour

# Phase 3
Prendre un acteur d'une file de tâches
Invoquer l'acteur

Fin étape d'ordonnancement
```

ALGORITHME 2 – *Description d'une étape d'ordonnancement.*

5.1.1 Implémentation pour STHORM

Les spécificités de l'architecture sur laquelle ont été menées les expérimentations – et de sa pile logicielle – rendent nécessaires un certain nombre d'adjonctions au modèle d'exécution et à l'ordonnanceur tels qu'ils ont été présentés précédemment. Il s'agit également de montrer ici un exemple concret d'implémentation de ces concepts.

Le premier raffinement proposé concerne la notion de contrôleur introduite par le modèle d'exécution. Pour le rendre compatible avec PEDF et faciliter sa mise en œuvre, il a été nécessaire de le décomposer en deux entités distinctes représentées par la figure 5.4: un prologue chargé de la collecte des paramètres d'entrée, de la reconfiguration du filtre et de son invocation; et un épilogue pour recueillir les paramètres produits par le filtre, effectuer le post-traitement et les transmettre aux acteurs destinataires.

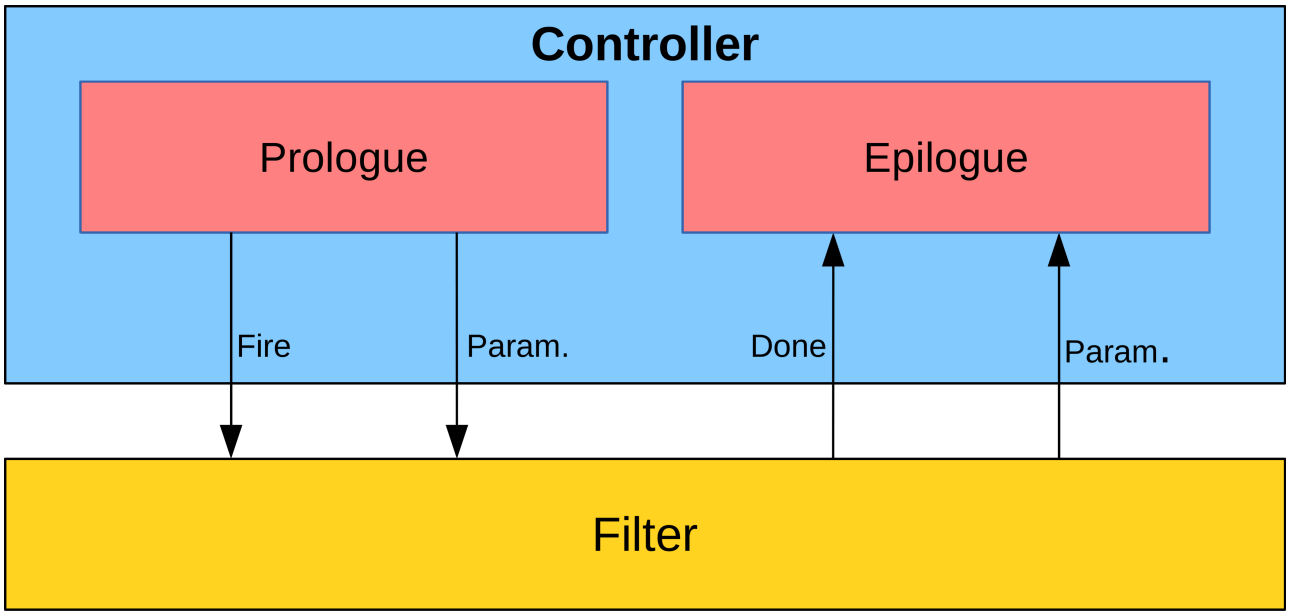


FIGURE 5.4 – *Décomposition d'un contrôleur en couple prologue-épilogue.*

Ainsi, du point de vue de l'ordonnanceur, l'invocation d'un acteur – ou, de façon équivalente, l'exécution d'une tâche – revient à l'appel de la fonction prologue correspondante. La fonction épilogue est quant à elle exécutée après acquittement du signal done par l'ordonnanceur. Ce dernier point contraste avec ce qui a été présenté jusqu'ici, puisqu'une invocation n'apparait alors plus comme une et indivisible du point de vue de l'ordonnanceur, mais comme une succession de deux opérations disjointes entre lesquelles celui-ci vient s'intercaler. Cet écart par rapport au modèle se justifie par un souci de simplification de l'implémentation : une solution à base d'interruptions était envisageable mais se serait révélée coûteuse et problématique à cause de la durée potentiellement longue de l'épilogue. La méthode retenue a l'avantage de l'efficacité, bien qu'elle soit susceptible de retarder légèrement la mise à disposition des paramètres de sortie le temps de l'acquittement par l'ordonnanceur du signal done, et, surtout, elle n'amoindrit en rien les garanties sur la sémantique.

Sur STHORM, l'implémentation est simplifiée par la présence de compteurs atomiques matériels, accessibles à la fois par les SWPE et les HWPE, qui peuvent être lus et décrémentés en une seule opération. La figure 5.5 illustre le procédé : lorsqu'un acteur achève une invocation, il émet le signal done en direction du compteur associé, ce qui a pour effet de l'incrémenter inconditionnellement ; l'acquittement se fait ensuite lors de la phase 1 de la prochaine étape d'ordonnancement par post-décrémentation, c'est-à-dire que la valeur retournée – 1 en cas de réussite, 0 en cas d'échec – est celle précédant la tentative de modification. Dans l'exemple de la figure 5.5, l'acteur A_1 a terminé son invocation, produit le paramètre P_1 et envoyé le signal done, que l'ordonnanceur a acquitté avant de mettre à jour l'état des paramètres (P_1 est disponible et P_2 en attente) ; l'acteur A_N nécessitant uniquement P_1 comme paramètre d'entrée, il est alors marqué prêt, tandis que A_1 reste inactif.

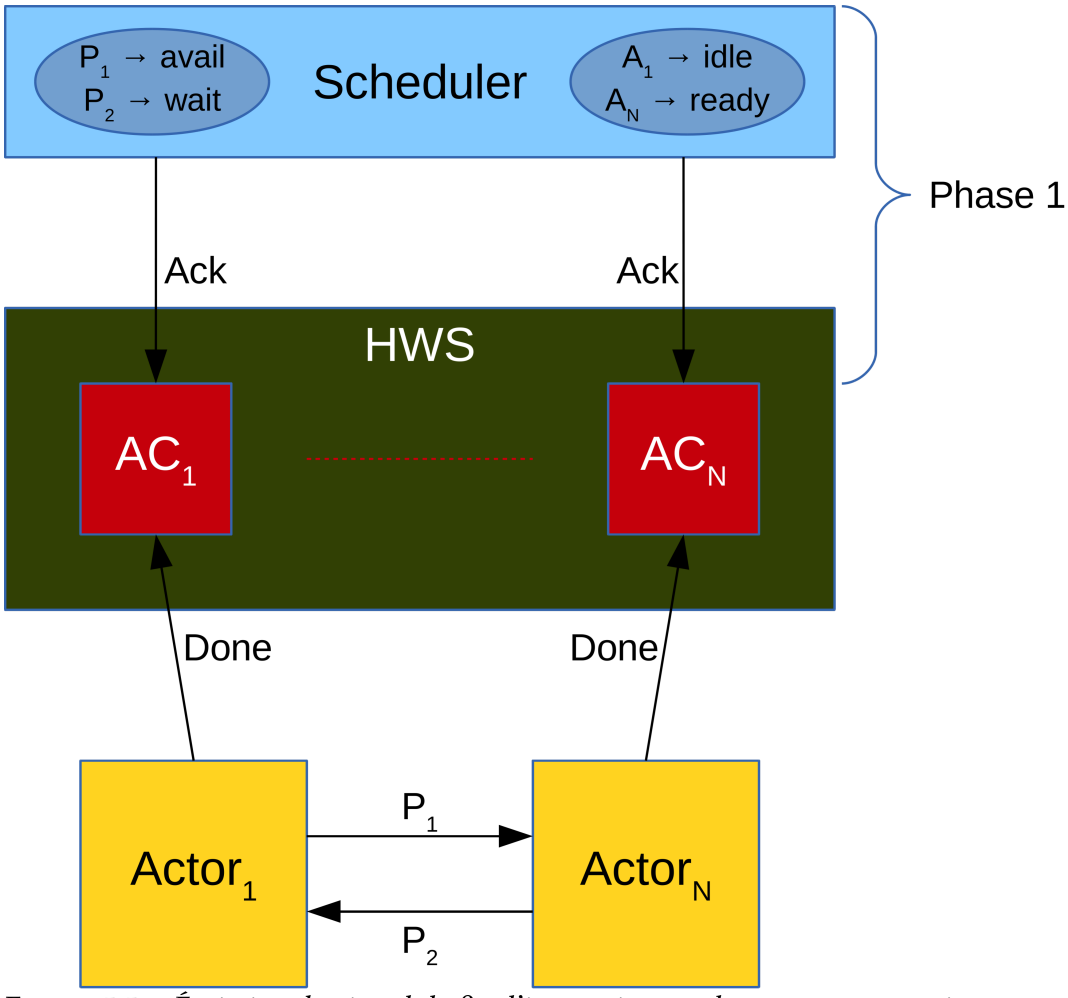


FIGURE 5.5 – *Émission du signal de fin d'invocation par les acteurs et acquittement par l'ordonnanceur* via *les compteurs matériels.*

Le recours aux compteurs atomiques permet de se prémunir contre les accès concurrents relatifs à l'acquittement des signaux done. En ce qui concerne l'ordonnancement des tâches prêtes à la phase 2, la lecture et la commutation de l'état peuvent se faire de manière indivisible grâce à une instruction spécifique qui verrouille le bus pendant toute la durée de l'opération.

Enfin, pour que l'ordonnanceur soit à même de mener à bien sa mission, l'application doit lui fournir un certain nombre d'informations indispensables, parmi lesquelles le nombre d'acteurs et, pour chacun d'eux, la liste des paramètres d'entrée et de sortie, et les pointeurs vers les fonctions prologues et épilogues. L'application a également la responsabilité d'allouer les files destinées à la transmission des paramètres, ainsi que les données devant être passées à l'ordonnanceur. Il revient à ce dernier de fournir des interfaces clairement définies à cet effet.

5.1.2 Politiques d'ordonnancement

Selon Parks [24], un ordonnanceur dynamique destiné aux réseaux de processus se doit de satisfaire à deux exigences: il doit garantir une exécution complète – c'est-à-dire qui n'aboutit pas à une impasse – et bornée – en termes d'accumulation de jetons dans les canaux de communication. Dans le cas général, pour un programme donné, l'existence d'une exécution en mémoire bornée n'est pas décidable; cependant, dans le cadre des applications visées ici, l'hypothèse est faite qu'il est toujours possible d'en trouver une. Il en résulte qu'une politique d'ordonnancement correcte

au sens du modèle d'exécution proposé ne doit pas introduire de blocage artificiel lié à la capacité limitée des files. Compte tenu du fait que leur taille est fixée à la compilation, sans possibilité de redimensionnement dynamique, et que les changements de contexte ne sont pas autorisés, il importe de s'assurer, dans la mesure du possible, pour chaque invocation, que celle-ci ne risque pas de mener à une impasse liée à une tentative de lecture depuis une file vide ou d'écriture dans un canal déjà plein.

Pour garantir la correction de l'exécution, les règles suivantes sont nécessaires. Dans le cas DPN, il est interdit d'invoquer un acteur si un paramètre d'entrée est indisponible (règle d'activation) ou si un paramètre de sortie ne peut être produit pour cause d'encombrement de la file de destination. De la sorte, aucun blocage n'est possible. Dans le cas KPN, si des paramètres indicatifs sont fournis par l'application, alors la règle précédente s'applique. Autrement, s'impose une approche conservatrice : l'invocation n'est permise que si des jetons sont présents en entrée et les files de sortie ne sont pas pleines – sauf si c'est le seul moyen de débloquer d'autres processus, c'est-à-dire si aucune règle d'activation n'est vérifiée et si l'invocation permet de consommer des jetons depuis une file pleine ou d'en produire dans un canal vide. Cette dernière règle n'apporte pas de garantie formelle, mais permet une minimisation du risque de blocage.

Il est donc manifeste que l'absence conjuguée de règles d'activation et de changements de contexte a de lourdes conséquences quant à l'ordonnançabilité du modèle d'exécution. Comme mentionné au chapitre 2, ce choix de conception trouve sa justification dans le constat que, pour de nombreuses applications dont on ne dispose que d'une implémentation séquentielle, l'effort que nécessiterait la mise en place de règles d'activation en bonne et due forme peut être conséquent. Pour certaines, il arrive même que la quantité de jetons consommés ne soit pas connue au moment de l'invocation. Les changements de contexte sont quant à eux proscrits du fait de considérations de plus bas niveau liées à l'efficacité de l'implémentation.

5.2 Filtres logiciels

La section précédente a présenté les aspects de l'implémentation du modèle d'exécution et de l'ordonnanceur indépendants de la nature de la cible – matérielle ou logicielle. Celle-ci aborde spécifiquement la question du support de l'implémentation logicielle des filtres.

Bien qu'initialement conçu dans l'idée de pouvoir cibler aussi bien le logiciel que le matériel en partant du même code écrit par l'utilisateur, PEDF n'a en pratique jamais bénéficié de cette possibilité : les contrôleurs ne peuvent être que logiciels et les filtres que matériels. Les développements menés au cours de cette thèse ont tenté d'y remédier partiellement. Dans le cas des contrôleurs, il est vrai que l'accélération matérielle un temps envisagée ne saurait être favorable que dans l'éventualité où au moins un sous-ensemble de l'ordonnancement serait statique – ou, à la rigueur, quasi-statique (cf. chapitre 2) – et, surtout, où les reconfigurations seraient prédictibles, ce qui exclut donc d'emblée les applications visées dans le cadre de cette étude. En revanche, le support des filtres logiciels est une fonctionnalité précieuse :

- l'effort et le temps de développement sont réduits ;
- le prototypage rapide basé sur l'analyse de performances s'en trouve facilité ;

- la flexibilité est accrue ;
- la correction de bogues à posteriori devient possible.

Pour la mettre en œuvre, un certain nombre de questions d'implémentation doivent nécessairement être soulevées, le modèle d'exécution l'ayant quant à lui prise en compte dès les premières étapes de sa conception.

En tête de ces questions figurent la gestion de la mémoire et le support des mécanismes de communication à flot de données. La première découle notamment de la limite à 256 ko imposée par l'implémentation de STHORM utilisée pour les expérimentations sur la capacité de la mémoire locale. Celle-ci contenant par défaut les piles des SWPE et les données des contrôleurs, dont la taille totale avoisinait déjà cette limite lorsque tous les filtres étaient en matériel, il n'était pas envisageable d'y adjoindre les données liées à leur implémentation logicielle. La mémoire de la fabrique, d'une capacité de 8 Mo mais qui n'est ordinairement pas utilisée par PEDF, a dû être réactivée et le mappage de l'application révisé comme suit :

- les piles dans la mémoire locale ;
- les données statiques dans la mémoire de la fabrique.

Par ailleurs, dans le cas de H.264, la combinaison de l'ordre de balayage des trames par ligne de macroblocs et de la prédiction spatiale – c'est-à-dire basée sur les macroblocs voisins déjà décodés – nécessitent pour certains acteurs de conserver en permanence une ligne complète d'échantillons décodés. Cette exigence, coûteuse en termes de mémoire, se traduit par la présence de lignes de délai rattachées aux acteurs concernés sous forme de simples files bouclées de grande capacité dans le cas matériel. Par souci du risque de saturation de la mémoire de la fabrique, les filtres logiciels utilisent également des lignes de délai matérielles dont l'autonomie induit une complexité accrue. Ainsi, un pseudo-acteur – implémenté sous forme de module PEDF – comprenant :

- un contrôleur ;
- un filtre d'entrée ;
- un filtre de sortie ;

remplit cet office. Le contenu de la ligne de délai est stocké dans le canal entre les deux filtres, et des files logicielles sont mises en place pour la communication avec le filtre logiciel de rattachement. Les deux filtres matériels jouent le rôle d'interfaces entre l'espace Von Neumann des SWPE et le monde des flux de données dans lequel évoluent les HWPE (cf. section 5.3). Ces pseudo-acteurs sont exécutés en continu : ils ne nécessitent pas d'ordonnancement.

La seconde question d'implémentation importante est relative à la gestion des communications à flot de données, sachant qu'il n'existe aucun support architectural à cet effet du côté des SWPE. En pratique, elle se décompose en deux éléments :

- des files modélisant les canaux de communication inter-acteurs ;
- une gestion des lectures et écritures au niveau des ports suivant la sémantique KPN.

Le premier reposant sur les mêmes fondements que les paramètres, le travail déjà effectué en la matière peut être mis à profit : l'implémentation des files étant générique – au sens polymorphique –, elle demeure réutilisable telle quelle. En revanche, les ports de paramètres obéissant quant à eux au modèle DPN, il n'y a dans ce cas pas de possibilité de réutilisation.

La solution retenue est l'attente active, à la fois pour les lectures bloquées sur des files d'entrée vides et les écritures arrêtées sur des canaux de sortie pleins. Il aurait éventuellement été possible d'endormir les SWPE en cas de blocage, puis de les réveiller par l'émission d'un évènement au moment opportun, afin d'en réduire la consommation énergétique, mais la réponse choisie a l'avantage de la réactivité et de l'efficacité. Enfin, des primitives simples pour la lecture et l'écriture depuis le code des filtres sont fournies pour remplacer le mécanisme original de PEDF basé sur une combinaison de macros, de tableaux et de surcharges d'opérateurs.

Par ailleurs, les changements de contexte, bien qu'ils soient déjà supportés par le logiciel système de PEDF et permettent de rendre l'exécution plus souple, n'ont pas été utilisés car ils ont un coût élevé :

- temporel : de l'ordre de 120 cycles de processeur sur STHORM ;
- spatial : une pile en mémoire locale par fil d'exécution.

Enfin, en ce qui concerne la mise en œuvre du modèle d'exécution, la fonction de travail d'un filtre logiciel est appelée directement depuis son contrôleur, ce qui implique que son code s'exécute dans le fil de l'ordonnanceur et que le contrôle n'est rendu à celui-ci qu'en fin d'invocation (mais néanmoins avant l'épilogue).

5.3 Communication matériel-logiciel

Comme il a été mentionné dans la section précédente, la particularité des architectures hybrides étudiées dans cette thèse – à savoir la cohabitation des modèles de calcul à flot de contrôle pour les SWPE et de données pour les HWPE – soulève des questions importantes en matière d'interfaces de communication. La réponse apportée par STHORM consiste à supporter les échanges de données bidirectionnels entre une mémoire quelconque (locale, fabrique ou externe) et un HWPE par l'entremise du DMA. Cette technique est efficace pour transférer de grandes quantités d'information en une seule fois depuis ou vers la mémoire externe en début et en fin de traitement. En revanche, pour des communications avec la mémoire locale à une granularité plus fine et moins prévisibles, cette approche est onéreuse. En effet, si le coût de configuration du DMA par un SWPE, à l'avance, en parallèle des calculs, est négligeable par rapport à la latence de la mémoire externe – de l'ordre de quelques centaines de cycles –, il en va tout autrement lorsqu'un HWPE doit intercaler des opérations de contrôle avec ses calculs et attendre fréquemment des données en provenance de la mémoire locale. Parmi les autres solutions architecturales, on peut citer l'adjonction d'extensions, soit côté HWPE soit côté SWPE, pour décomposer les flux de données en opérations d'écriture élémentaires dans un sens, et combiner les résultats de lectures successives en paquets dans l'autre. Cette dernière technique n'a pas été implémentée dans STHORM mais elle représente vraisemblablement l'option la plus efficace.

En l'absence d'un tel support, la technique retenue pour l'implémentation des communications entre filtres logiciels et matériels repose sur le recours massif à une fonctionnalité présente dans le

simulateur mais absente de la plateforme réelle. Ainsi, les accès *détournés* des HWPE permettent de simuler des lectures et écritures de blocs de taille arbitraire dans l'ensemble du plan mémoire, simulant ainsi grossièrement le mécanisme décrit au paragraphe précédent. Bien que cette possibilité soit commode pour le transfert de grandes quantités de données contigües, elle rend néanmoins l'implémentation des mécanismes de synchronisation complexe. Il est en effet nécessaire, après allocation des files (tampons et métadonnées) par les SWPE dans la mémoire de la fabrique, d'indiquer aux HWPE leurs adresses sous forme d'attributs. Par la suite, chaque transfert de jeton dans une file logicielle depuis un HWPE se traduit par cinq accès détournés : deux pour la lecture des compteurs, un pour la lecture de l'adresse de base du tampon, un pour le transfert proprement dit et un pour la mise à jour du compteur approprié.

Enfin, un cas particulier de communications liées aux filtres logiciels doit être envisagé : il s'agit des transferts entre mémoires externe et locale orchestrés par le DMA. Bien que le support pour ce type d'opérations préexistât dans PEDF, il n'avait jamais été utilisé pour des applications réelles. En effet, le DMA ne dispose pas de support pour les mécanismes à flot de données dans la mémoire globale : il est seulement capable de transférer le contenu de segments de mémoire à adresses fixes. Il a donc été nécessaire d'introduire des tampons intermédiaires et de rajouter des synchronisations pour garantir la correction des communications.

5.4 Assignation des acteurs aux ressources matérielles

Bien qu'il soit souhaitable de disposer d'un degré de liberté maximal quant à l'ordonnancement des applications très dynamiques, ce qui implique notamment de pouvoir décider, jusqu'à l'invocation, du placement de chaque acteur sur les ressources matérielles disponibles, les contraintes architecturales de STHORM et les limitations théoriques du modèle imposent des restrictions sur cette latitude. Ainsi, s'il est tout à fait possible de choisir, pour les filtres logiciels, le SWPE assigné à chaque invocation à des fins d'équilibrage de charge et de réactivité, il n'en va pas de même pour les filtres matériels. Le placement de ces derniers est en effet contraint à la fois par le modèle de la plateforme et par le modèle de programmation de PEDF qui imposent tous deux une assignation statique aux HWPE, ce qui résulte notamment de la très haute spécialisation de ces accélérateurs[30]. Il serait néanmoins envisageable de prendre cette décision au chargement de l'application, ce qui permettrait par exemple d'effectuer au préalable un calibrage de la plateforme et de prendre en compte des informations telles que le niveau de la batterie pour un appareil portable : si celle-ci est faible, on préfèrera l'implémentation la moins consommatrice, même si la qualité de restitution s'en ressent.

En deuxième lieu, une autre possibilité attrayante aurait été de choisir à l'exécution entre implémentation matérielle et logicielle. Cela est impossible pour de nombreuses raisons dont les principales sont exposées ici. La première d'entre elles concerne l'état d'un acteur, c'est-à-dire les données privées qu'il conserve d'une invocation sur l'autre : si plusieurs implémentations coexistent, alors il devient nécessaire d'assurer la cohérence de cet état entre elles. La solution la plus simple serait la duplication, c'est-à-dire que chaque instance maintiendrait son propre état, mais un mécanisme supplémentaire serait nécessaire et il n'est pas clair dans quelle mesure l'ensemble apporterait un gain net. Une autre possibilité serait le partage de l'état entre les différentes

30 C'est d'ailleurs à l'aune de cette observation que l'on prend conscience du bénéfice relatif apporté par la flexibilité des SWPE.

instances, mais le risque d'accès concurrents devrait alors être circonscrit par le biais de verrous et de files supplémentaires, ce que les modèles à flot de données entendent précisément éviter et qui rendrait l'ensemble inutilement complexe. Enfin, on pourrait transformer le graphe de manière à ce que les acteurs soient purement fonctionnels, c'est-à-dire que les sorties soient déterminées exclusivement par les entrées, et faire transiter les états sous forme de paramètres : au coût, certes modeste, que représenterait la gestion d'un paramètre supplémentaire pour chaque acteur s'ajoute celui, nettement plus lourd, lié au transfert proprement dit de quantités de données éventuellement conséquentes. Cette dernière solution est vraisemblablement la plus évidente à implémenter mais il s'agit aussi certainement de la plus coûteuse à l'exécution, ce qui ne serait pas acceptable vu les contraintes en matière de réactivité et d'efficacité auxquelles le modèle d'exécution et son implémentation sont soumis.

Par ailleurs, outre les questions afférentes à l'état des acteurs, la cohabitation de plusieurs implémentations d'un même acteur et le choix à l'exécution de l'assignation soulèvent un certain nombre de problèmes. Parmi ceux-ci, le choix de la file de sortie apparaît comme l'un des plus prégnants : la décision de l'assignation d'un successeur doit être connue avant que l'acteur en cours d'invocation ne commence à produire des jetons, si l'on s'en tient au modèle d'exécution retenu ; autrement, ceux-ci devraient être stockés dans un tampon intermédiaire dans l'attente que le placement du consommateur soit connu. Dans le premier cas, cela implique de prendre les décisions d'allocation très en avance, et donc que celles-ci se révèlent moins judicieuses au moment de l'invocation ; dans le second, il est clair que l'accroissement de l'empreinte mémorielle induite par la multiplication des tampons n'est pas soutenable dans des environnements aussi contraints que ceux visés.

Les problèmes soulevés dans les paragraphes précédents se révèlent particulièrement aigus lorsqu'il est question d'ordonnancement dynamique. En effet, la contrainte sur l'ordre d'exécution des invocations d'un même acteur conjuguée à celle sur les états laissent peu de latitude à l'ordonnanceur. Il aurait par exemple été souhaitable d'invoquer certains acteurs dans le désordre de manière à amortir la latence liée aux différences de temps d'exécution d'une invocation à l'autre. Les contraintes imposées par le modèle d'exécution sur l'ordonnancement sont donc fortes mais elles sont justifiées par la nécessité de garder le logiciel système – et en particulier l'ordonnanceur – simples et légers à la fois aux points de vue spatial et temporel.

5.5 Intégration dans PEDF et adaptation des applications

Les nombreux ajouts, modifications et adaptations induits par la mise en œuvre du modèle d'exécution et l'introduction de l'ordonnanceur présentés précédemment dans PEDF et les applications cibles se sont traduits par un travail de développement conséquent dont cette section tente de dessiner les grandes lignes.

Les modules de la pile logicielle de PEDF touchés sont, par ordre d'importance décroissante :

- le logiciel système ;
- l'extension du compilateur MIND ;
- la couche d'abstraction matérielle ;

– l'interface entre la plateforme et les modèles des filtres.

Le logiciel système concentre l'essentiel des ajouts liés :

– au modèle d'exécution : structures de données pour la description de l'application, des acteurs et des paramètres, implémentation générique des files sous forme de tampons circulaires et des ports de paramètres ;

– aux filtres logiciels : description et implémentation des ports de données, génération de code pour l'allocation des files de données, implémentation de primitives de transfert, initialisation des HWPE pour les files matériel-logiciel ;

– à l'ordonnanceur : boucle principale, commutation des acteurs prêts, détection de la fin d'exécution et transmission au contrôleur de la fabrique, implémentation des files de tâches.

La couche d'abstraction matérielle de PEDF a dû être augmentée de primitives pour :

– l'invocation sélective de filtres matériels ;

– l'émission et l'acquittement des signaux de fin d'invocation ;

– le signalement des fins d'exécution par chaque filtre.

Les modifications apportées à la chaine de compilation sont liées au mappage :

– retrait de l'assignation statique des contrôleurs aux SWPE (sauf pour le macro-acteur `front`) ;

– remplacement des appels spécifiques aux fonctions de travail des contrôleurs par l'appel inconditionnel de la boucle d'ordonnancement principale sur tous les SWPE ;

– synchronisation de tous les processeurs avec le contrôleur de la fabrique en début et en fin d'exécution ;

– configuration des compteurs atomiques pour la gestion des fins d'invocation et d'exécution ;

– accès aux méta-données de PEDF depuis la description de chaque acteur ;

– invocation inconditionnelle des lignes de délai qui ne sont pas soumises à ordonnancement.

Enfin, l'interface de bas niveau des filtres matériels s'est vue dotée de l'accès aux files de données en mémoire locale depuis les HWPE, ainsi que de la modélisation des délais au niveau du code de l'utilisateur.

En outre, les applications doivent se conformer à l'interface définie par l'ordonnanceur pour fournir les informations requises concernant notamment les acteurs et les paramètres (cf. section 5.1.1). Un certain nombre de routines ont également dû être écrites pour satisfaire au modèle d'exécution : constructeurs, prologues et épilogues. Il incombe finalement à l'application de mettre à jour l'état des acteurs, d'ajuster les paramètres d'entrée de chacun et d'appeler la primitive d'invocation depuis les fonctions de travail des contrôleurs.

5.6 Conclusion

Ce chapitre a décrit une architecture d'ordonnanceur adaptée au modèle d'exécution proposé précédemment, son implémentation sur STHORM, ainsi que les autres contributions à la pile logicielle de PEDF. Ces développements, qui représentent une part importante de l'ensemble des travaux menés au cours de cette thèse, démontrent la faisabilité de l'approche proposée. D'abord, du point de vue de la plateforme, l'hybridité est prise en compte par la mise en place de mécanismes spécifiques permettant des transferts facilités entre SWPE et HWPE. Ensuite, du point de vue de l'application, le fort dynamisme est capturé par une structure d'ordonnanceur distribuée réactive à trois phases et une gestion dynamique adaptée des invocations. Ce dynamisme au niveau du logiciel système se paye au prix d'un risque d'impasse qui ne peut pas toujours être évité. Ce compromis est jugé satisfaisant dans la mesure où les solutions alternatives sont coûteuses et en temps humain et en temps de processeur.

Le chapitre suivant évalue le prototype résultant par le biais d'expérimentations sur les applications décrites précédemment.

Chapitre 6. Expérimentations

Les précédents chapitres ont examiné les concepts théoriques et pratiques permettant la mise en œuvre efficace d'applications à flux de données sur les architectures embarquées hybrides. Plus précisément : le chapitre 2 a proposé un modèle d'exécution adapté, le chapitre 4 a décrit les techniques de développement et les applications utilisées, et le chapitre 5 a abordé l'implémentation du support logiciel requis. Le présent chapitre a quant à lui pour objet d'attester par l'expérience de la validité de l'approche et d'évaluer quantitativement le gains nets apportés. La première section détaille le protocole expérimental et la seconde expose les résultats.

6.1 Protocole expérimental

Les expériences présentées ici visent à illustrer l'utilité du modèle d'exécution proposé et notamment des paramètres indicatifs. La première sous-section décrit leur utilisation, la seconde expose les métriques utilisées et la dernière détaille les politiques d'ordonnancement retenues pour les expérimentations.

6.1.1 Paramètres indicatifs

Comme il a été vu au chapitre 2, le modèle d'exécution introduit une nouvelle classe de paramètres qualifiés d'*indicatifs*. Ces paramètres, facultatifs au sens où ils ne sont pas nécessaires à l'exécution correcte de l'application, peuvent néanmoins apporter des informations potentiellement utiles à l'ordonnanceur. Dans ce qui suit, les renseignements délivrés seront de deux types : indications sur la quantité de jetons transférés[31] à chaque invocation et indications sur la durée de chaque invocation. Ces informations pourraient se révéler précieuses dans le domaine KPN où aucune indication sur les débits n'est requise par le modèle, ce qui a pour conséquence d'introduire un risque de blocage (cf. chapitre 5). L'apport des paramètres indicatifs relatifs aux quantités de jetons permettrait alors d'amoindrir ce risque, comme le montre la section suivante. Ceux relatifs à la durée des invocations permettraient d'amortir la latence, sous réserve que les files interacteurs soient correctement dimensionnées, par exemple en ordonnançant en priorité les invocations courtes.

6.1.2 Flot moyen

Cette section a pour objet de justifier le recours aux paramètres indicatifs. Pour ce faire, on montre que l'optimisation de l'utilisation des ressources passe par la comparaison des durées des invocations, et donc la connaissance relative de celles-ci. Soient A, B et C trois acteurs formant le graphe à flot de données représenté par la figure 6.1.

31 On ne parlera pas ici de *débits* car ce terme s'emploie uniquement pour désigner le nombre exact de jetons transférés par chaque port. Les informations apportées par les paramètres indicatifs peuvent être partielles voire approximatives.

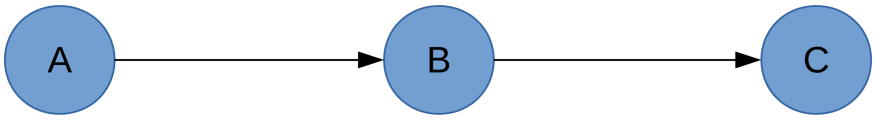


FIGURE 6.1 – *Graphe de trois acteurs pour illustrer le flot moyen.*

On note A_i la i-ème invocation de l'acteur A. On définit $d_a(A_i)$ la date d'activation de A_i c'est-à-dire l'instant à partir duquel les i–1 premières invocations de A sont achevées et la règle d'activation de A (ou, de manière équivalente, sa dépendance dans le graphe) est satisfaite. On définit $d_f(A_i)$ la date de fin de A_i. On définit le flot moyen, noté f, comme la fonction qui à une invocation associe la différence entre sa date de fin et sa date d'activation : $f(A_i) = d_f(A_i) - d_a(A_i)$. Cette métrique permet de quantifier, pour chaque invocation, à la fois le temps de processeur utilisé et la durée minimale d'occupation des files inter-acteurs par les jetons qu'elle consomme ; en d'autres termes, elle mesure la durée d'utilisation des ressources à quantité de travail donné. Optimiser l'utilisation des ressources revient donc à minimiser f.

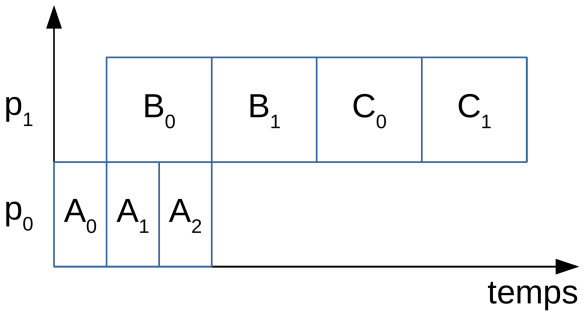


FIGURE 6.2 – *Possibilité d'ordonnancement O_1.*

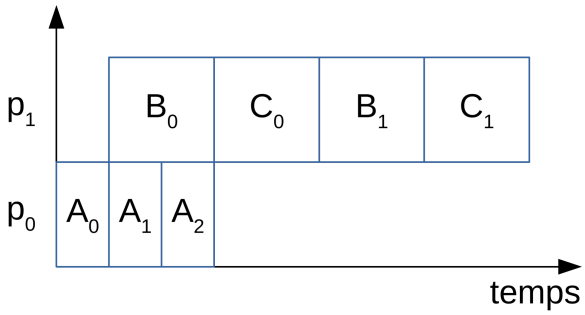


FIGURE 6.3 – *Possibilité d'ordonnancement O_2.*

On suppose que l'environnement de calcul comprend deux processeurs p_0 et p_1 et que les trois acteurs sont assignés statiquement : A est seul sur p_0, B et C partagent p_1. On souhaite comparer à l'aune du flot moyen deux possibilités d'ordonnancement sur p_1 illustrées par les figures 6.2 et 6.3 :

- $O_1 : B_0\ B_1\ C_0\ C_1$;
- $O_2 : B_0\ C_0\ B_1\ C_1$.

Soit F_i le flot moyen total de l'ordonnancement O_i. Les durées des invocations sont notées en minuscule, ainsi a_0 représente la durée de l'invocation A_0. On a alors :

$$\begin{aligned} F_1 &= f(B_0) + f(B_1) + f(C_0) + f(C_1) \\ &= (a_0 + b_0 - (a_0)) + (a_0 + b_0 + b_1 - (a_0 + b_0)) + (a_0 + b_0 + b_1 + c_0 - (a_0 + b_0)) + (a_0 + b_0 + b_1 + c_0 + c_1 - (a_0 + b_0 + b_1 + c_0)) \\ &= 2b_1 + c_0\ , \\ F_2 &= f(B_0) + f(C_0) + f(B_1) + f(C_1) \\ &= (a_0 + b_0 - (a_0)) + (a_0 + b_0 + c_0 - (a_0 + b_0)) + (a_0 + b_0 + c_0 + b_1 - (a_0 + b_0)) + (a_0 + b_0 + c_0 + b_1 + c_1 - (a_0 + b_0 + c_0 + b_1)) \\ &= b_1 + 2c_0\ . \end{aligned}$$

D'où :

$$\begin{aligned} F_2 - F_1 &= c_0 - b_1 \\ &> 0 \text{ si } c_0 > b_1 \\ &< 0 \text{ si } c_0 < b_1 . \end{aligned}$$

Il en résulte que O_1 est meilleur si et seulement si C_0 a une durée supérieure à B_1. On en conclut que le choix du meilleur ordonnancement dépend des durées d'invocation de B et de C: on doit ordonnancer le plus court en premier. Ce résultat justifie le recours aux paramètres indicatifs relatifs aux durées d'invocation.

6.1.3 Stratégies d'ordonnancement

Dans le but d'évaluer la structure de l'ordonnanceur proposée au chapitre 5 ainsi que l'impact des paramètres indicatifs, plusieurs politiques d'ordonnancement ont été sélectionnées en vue d'être comparées:

- S_0: ordonnancer alternativement tous les acteurs de façon à ce que chacun ait la même chance d'être invoqué;
- S_1: ordonnancer en priorité les acteurs en amont dans le graphe, puis descendre vers les acteurs en aval en suivant l'ordre topologique au fur et à mesure que les files inter-acteurs se remplissent;
- S_2: ordonnancer en priorité les acteurs en aval dans le graphe de manière à ce que les jetons produits par les acteurs en amont soient consommés dès que possible;
- S_3: même chose que S_1 mais seulement pour les invocations courtes.

S_0 est à la fois la stratégie la plus simple à implémenter et la plus juste au sens où tous les acteurs ont autant de chance d'être invoqués. Les trois autres stratégies vont au contraire privilégier certains acteurs, voire certaines invocations, dans le but de compenser les variations de durée d'exécution. En cherchant à tirer parti au maximum de la capacité des files inter-acteurs, la stratégie S_1 tente d'amortir la latence introduite par une succession d'invocations de durées inégales en permettant aux plus rapides de ne pas être bloquées par les plus lentes. La stratégie S_2 fait le choix inverse et tente de minimiser l'occupation des files en favorisant la consommation des jetons dès leur production. Enfin, la stratégie S_3 s'appuie sur le résultat de la section précédente en choisissant d'optimiser l'utilisation des ressources et la cadence d'exécution sur la base du flot moyen: les invocations courtes d'acteurs en amont sont ordonnancées en priorité.

6.1.4 Applications

Les expériences ont été menées sur les deux applications décrites au chapitre 4: l'algorithme d'amélioration de la qualité d'image TNR et le décodeur vidéo H.264. Dans le cas du TNR, tous les filtres sont matériels; pour H.264, `ipred` et `ipf` sont en logiciel et les autres filtres en matériel. La durée de ces derniers a été modélisée *via* la technique d'introduction de délais décrite au chapitre 4, la composante aléatoire étant tirée sur une loi bêta de paramètres (2, 2) et le facteur de délai constant étant fixé à 100.

Dans H.264, les commandes envoyées par `front` à `ipred` et `ipf` (cf. chapitre 4) sont utilisées comme paramètres indicatifs concernant les durées d'invocation et les jetons transférés, comme le montre le tableau 7.

Commande	Durée de l'invocation d'`ipred`	Données transférées par `ipred` et `ipf`
`picture_cmd`	courte	non
`slice_cmd`	courte	non
`intra_long_cmd`	moyenne	oui
`inter_long_cmd`	longue	oui

TABLEAU 7 – *Informations portées par les paramètres indicatifs dans H.264.*

Il est à noter que les informations fournies sont essentiellement d'ordre qualitatif: les indications sur les durées permettent de comparer les invocations mais, prises séparément, ne sont d'aucun secours; les renseignements sur les données indiquent juste si une invocation transfère des jetons ou non, mais ne précisent pas leur nombre. Si les débits exacts étaient connus, alors l'acteur pourrait intégralement être décrit en DPN, mais ce n'est pas le cas, en général, dans H.264.

6.1.5 Simulation et mesures

Comme indiqué au chapitre 4, toutes les expériences ont été conduites sur la plateforme TLM de STHORM car il s'agit du simulateur offrant le meilleur compromis entre durée de simulation, facilité de mise en œuvre et précision du modèle. En particulier, l'ISS est doté d'outils appréciables pour le débogage et, en principe, le profilage de code (points chauds, goulets d'étranglement, etc.) intrusif – par annotation des sources puis analyse *post-mortem* – ou non intrusif – par instrumentation du simulateur. Mais, du fait d'un bogue lié à l'implémentation du chargement dynamique, aucune des fonctionnalités de profilage n'était disponible[32].

Par ailleurs, STHORM dispose, au sein de chaque groupe de calcul, d'un jeu de compteurs de cycles ou de temps, mais qui ne sont pas utilisables en TLM. Restent les compteurs intégrés aux SWPE, au nombre de deux par cœur, qui, dans le modèle de l'ISS, estiment le nombre de cycles à partir du nombre d'instructions. Enfin, l'environnement SystemC fournit une mesure de temps approximative liée à la modélisation des différents blocs matériels. Les relevés dont fait état la section suivante sont tous issus soit des compteurs de cycles des SWPE, soit du temps SystemC.

6.2 Résultats

La première sous-section présente les résultats obtenus sur l'application TNR et la seconde sur le décodeur H.264.

32 Le bogue a été dûment signalé au cours de la thèse mais n'a jamais été corrigé.

6.2.1 TNR

Cette application étant statique – au sens où elle nécessite peu de reconfigurations, toutes prédictibles, et qui n'affectent ni la structure du graphe ni les débits –, elle fait figure de bon candidat pour l'évaluation du coût brut de la gestion dynamique des acteurs et de la structure d'ordonnanceur proposées au chapitre précédent. L'expérience a consisté à exécuter et à comparer deux versions différentes du TNR: l'une (appelée *reference PEDF*) issue de l'implémentation PEDF d'origine, monolithique, avec un ordonnancement manuel optimisé, l'autre (nommée *scheduled PEDF*) étant celle présentée au chapitre 4, adaptée au modèle d'exécution et utilisant les contributions du chapitre 5. La première est par construction plus efficace par le simple fait qu'elle ne contient qu'un seul module et bénéficie donc d'un contrôleur centralisé et simplifié. Dans la deuxième, l'ordonnanceur utilise une politique simple qui ne tient pas compte de l'état des files de données et ordonnance les acteurs prêts par ordre topologique. De manière à modéliser la variabilité des temps de calcul, tous les filtres sont dotés de délais, comme indiqué à la section 6.1.4, dont les durées de référence sont rassemblées dans le tableau 8.

`fading`	8 µs
`spaY`	4 µs
`tempY`	3 µs
`tempUV`	3 µs
`estimateNoise`	5 µs
`motionDetect`	2 µs

TABLEAU 8 – *Temps de calcul de référence pour les filtres du TNR.*

On mesure, pour chaque version, les durées totales d'exécution en temps SystemC pour les valeurs de facteur de délai constant (CDF)[33] suivantes: 1, 2, 5, 10, 20, 50 et 80. La figure 6.4 illustre les résultats.

33 Comme énoncé au chapitre 4, CDF représente le coefficient multiplicatif constant des durées de référence des filtres: plus CDF est grand, plus la durée totale est élevée.

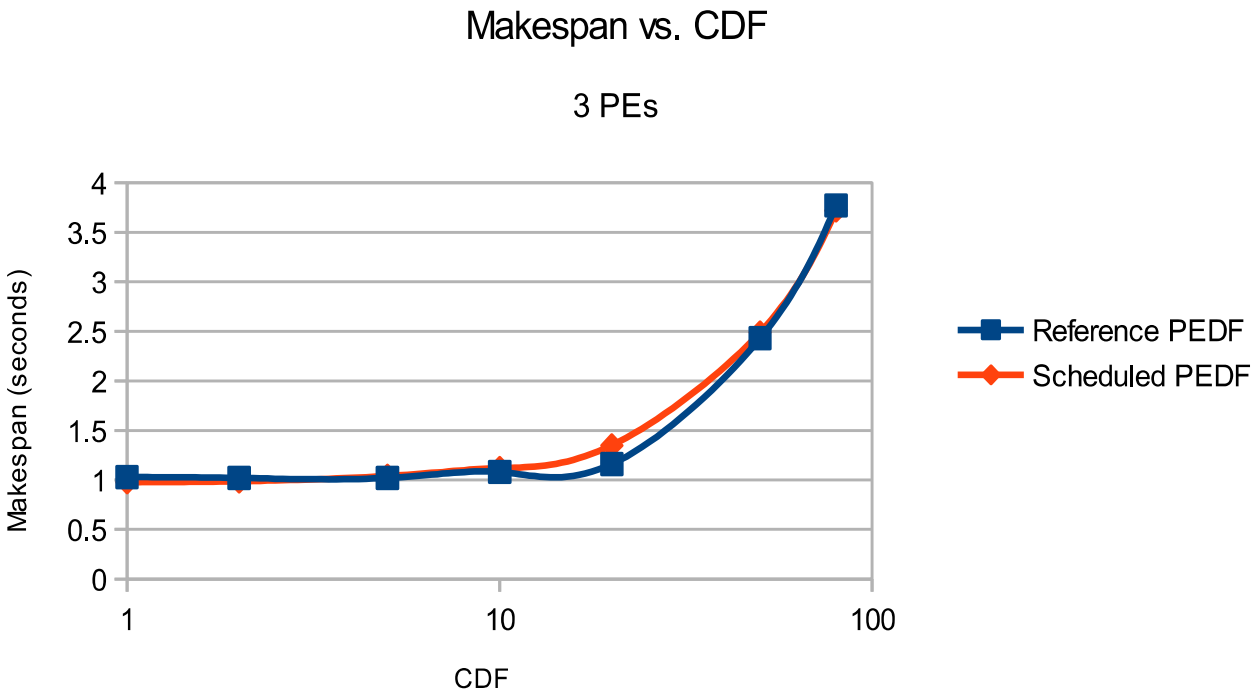


FIGURE 6.4 – *Durée totale d'exécution en fonction du facteur de délai constant.*

Pour les valeurs basses de CDF, les performances des deux implémentations sont équivalentes, avec un avantage pour l'ordonnanceur de l'ordre de 4 %. En revanche, quand le facteur de délai augmente, l'écart se creuse jusqu'à atteindre 16 % en faveur de la référence pour CDF = 20, puis redevient à nouveau faible (de l'ordre de 1 %) à partir de CDF = 50. On peut donc distinguer trois comportements distincts en fonction de la variabilité de la durée des invocations. Pour une variabilité faible (CDF $\in [1; 5]$), les temps de calcul sont du même ordre de grandeur que le temps passé par les SWPE à effectuer les opérations de contrôle incompressibles et l'ordonnancement : l'ordonnanceur est aussi bon que la référence. Pour une variabilité moyenne, les temps de calcul commencent à dominer d'où une augmentation rapide de la durée totale d'exécution : la référence est meilleure que l'ordonnanceur. Enfin, pour une variabilité élevée, tout le contrôle est négligeable par rapport au calcul : l'ordonnanceur obtient à nouveau des résultats comparables à la référence. En somme, dans le pire des cas, le modèle d'exécution, l'ordonnanceur et la gestion dynamique sont 16 % moins bons que l'implémentation de référence, statique et optimisée à la main, et, dans la plupart des cas, les performances sont comparables.

Les mesures précédentes ayant permis d'évaluer le coût temporel, il reste à en estimer l'aspect spatial, c'est-à-dire l'empreinte mémorielle. Pour ce faire, le tableau 9 compare la taille originale de différentes sections du logiciel système de PEDF et celle après l'implémentation des contributions présentées au chapitre 5. De manière prévisible, l'essentiel du surcoût réside dans la taille du code (section `.text`) qui passe de 4,7 ko à 7,6 ko, ce qui reste raisonnable comparé à celle de l'application (23 ko) et à la capacité des caches d'instructions (32 ko). Le coût des données est quant à lui inférieur à 100 octets. On remarquera par ailleurs le coût élevé de l'édition dynamique des liens (sections `*.dyn*` et `.hash`) qui dépasse à lui seul celui du code, aussi bien dans l'implémentation originale que dans la nouvelle, ce qui justifie les réserves émises au chapitre 1 (cf. p. 29). Il en résulte que le coût spatial est faible si l'on excepte la composante relative à l'édition des liens dynamique.

Section	Taille originale (octets)	Nouvelle taille (octets)	Surcoût	
`.text`	4744	7626	2882	61 %
`.dynsym`	2928	3552	624	21 %
`.dynstr`	1712	2027	315	18 %
`.rela.dyn`	936	1248	312	33 %
`.hash`	884	1124	240	27 %

TABLEAU 9 – *Estimation du surcoût spatial de l'implémentation du modèle d'exécution, de l'ordonnanceur et de la gestion dynamique dans le logiciel système.*

6.2.2 H.264

Contrairement au TNR, le décodeur H.264 utilisé pour ces expérimentations est une application hautement dynamique dont les variations et reconfigurations internes sont difficilement prévisibles. Il s'agit donc d'un bon cas d'étude pour évaluer l'adéquation du modèle d'exécution, l'efficacité de la gestion dynamique et le potentiel des paramètres indicatifs à capturer la variabilité. Les résultats présentés dans cette section découlent de mesures réalisées sur un nombre variable de trames d'une même séquence vidéo comprenant neuf lignes de onze macroblocs. Pour chaque expérience, les paramètres sont :

- la stratégie d'ordonnancement S_n avec $n \in [0\,;\,3]$;
- la taille des files logicielles inter-acteurs $F_{n'}$ avec $n' \in [1\,;\,3]$.

La convention de nommage sera donc S*n*F*n'*, de telle sorte que S0F3 désigne l'exécution utilisant la politique d'ordonnancement S0 avec des files de taille 3. Les expériences présentées ici s'intéressent à la *période* de décodage, c'est-à-dire au temps écoulé entre la livraison de deux éléments successifs de même type, macrobloc ou trame complète. En effet, dans un contexte de décodage vidéo en temps réel, le critère primordial est la capacité du système à délivrer sa sortie à une cadence suffisante pour assurer une qualité de service optimale ; en d'autres termes, il s'agit de faire en sorte que l'expérience de l'utilisateur ne soit pas dégradée par une période de décodage trop élevée qui se traduirait par des artéfacts voire des gels de l'image.

Les figures 6.5 et 6.6 illustrent les variations de durée d'invocation liées aux différentes commandes (cf. section 6.1.4). Il s'agit de mesures d'intervalle de temps entre deux macroblocs décodés successifs réalisées sur une même image de la séquence vidéo. Les macroblocs intra-codés apparaissent en violet et ceux inter-codés en jaune-orangé car ces derniers sont en moyenne quatre fois plus longs à traiter au niveau d'`ipred`. Si l'on compare S1F3 (fig. 6.5) et S2F3 (fig. 6.6), on observe un décalage de deux macroblocs dans l'ordre de balayage. Le décalage vaut en général f–1 où f est la taille des files. Ainsi pour f=3, le décalage vaut 2. Ce phénomène reflète la symétrie entre S1 et S2 : la première privilégie l'exécution des acteurs en amont et le remplissage des files, tandis que la seconde favorise les acteurs en aval et le vidage des files.

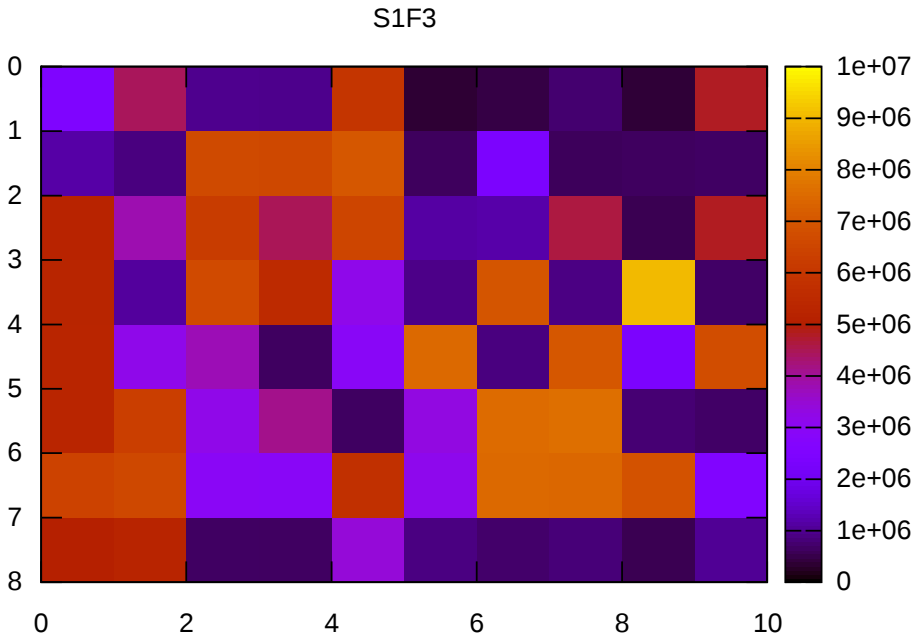


FIGURE 6.5 – *Représentation 2D d'une trame décomposée en macroblocs. La couleur indique le temps écoulé depuis le précédent macrobloc décodé.*

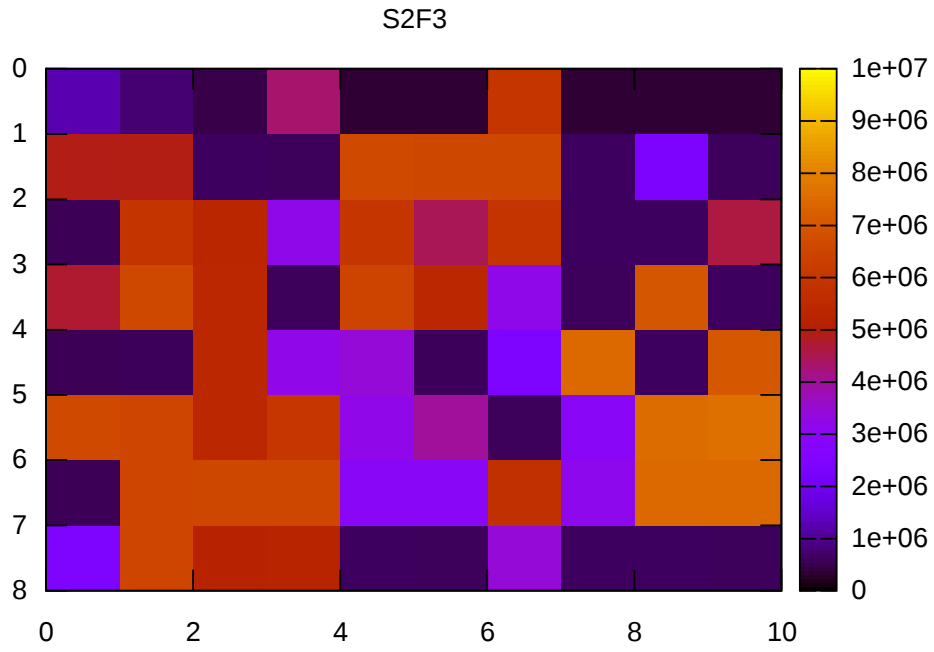


FIGURE 6.6 – *Représentation 2D d'une trame décomposée en macroblocs. La couleur indique le temps écoulé depuis le précédent macrobloc décodé.*

Les figures 6.7 à 6.9 illustrent les variations de période d'une image à l'autre en fonction du nombre d'alternances entre macroblocs inter-codés et intra-codés au sein d'une même trame. Comme plusieurs images peuvent présenter le même nombre d'alternances – ce qui se traduirait graphiquement par un nombre variable de points pour une même abscissse –, les résultats bruts ont été moyennés pour avoir au maximum un point par abscisse, et les écarts-types ont été matérialisés par des barres d'erreur[34]. La stratégie S0 est prise comme référence et les résultats des autres politiques sont normalisés par rapport à ceux de S0. L'ensemble est représenté dans un repère semi-logarithmique.

34 L'absence de barre d'erreur signifie qu'il n'y avait qu'une seule valeur pour cette abscisse.

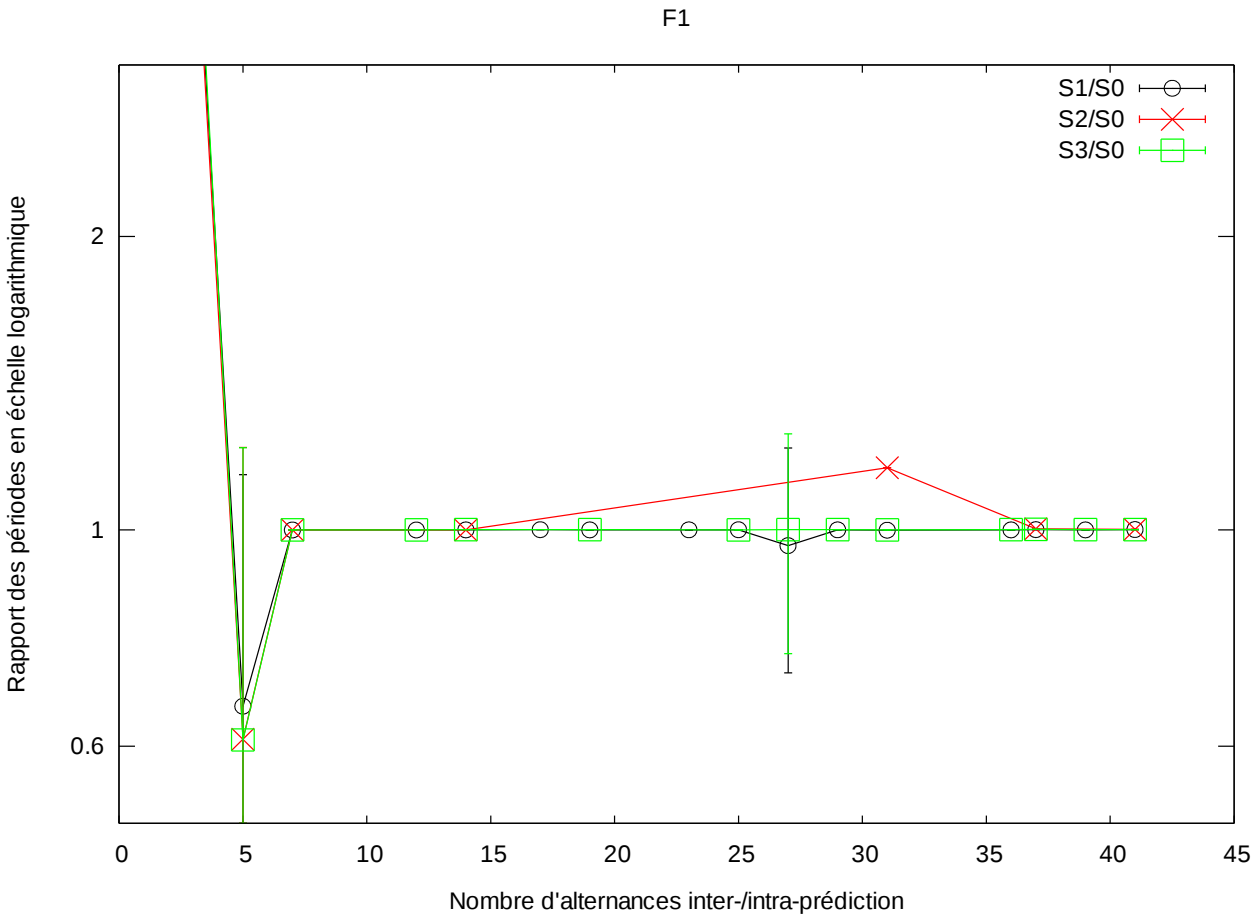


FIGURE 6.7 – *Rapports des périodes entre deux images décodées successives en fonction du nombre d'alternances entre inter- et intra-prédiction. Taille des files inter-acteurs : 1.*

La première observation concerne l'évolution lorsque la taille des files croît. Il est clair que, lorsque celle-ci est minimale, la marge de manœuvre de chaque politique est très limitée et les décisions prises aboutissent pour l'essentiel au même résultat. L'agrandissement des files a, en revanche, un impact net sur la capacité des différentes stratégies à peser réellement sur l'exécution. Il apparaît ainsi que les périodes mesurées commencent à s'écarter les unes des autres pour F2, et la tendance se confirme avec F3. Cela valide l'intuition selon laquelle l'augmentation de la taille des files permet de tirer parti des variations de latence de décodage au sein du pipeline.

Dans un second temps, il convient de s'attarder sur les différences entre les politiques d'ordonnancement. La première remarque qui s'impose est la supériorité incontestable de S0 en l'absence d'alternance, c'est-à-dire pour les trames codées intégralement avec le même type de prédiction : la période est environ cinq fois plus longue pour toutes les autres stratégies. En dehors de ce cas particulier, les rapports sont tous compris entre 0,8 et 1,4 pour F2 ou entre 0,6 et 2 pour F3. Dans le détail, dans le cas de F2, S2 et S3 ont des performances sensiblement identiques, tandis que S1 accuse un retard de l'ordre de 5 %. Le cas médian se situe aux alentours de 1,00 pour les deux premiers et de 1,04 pour S1.

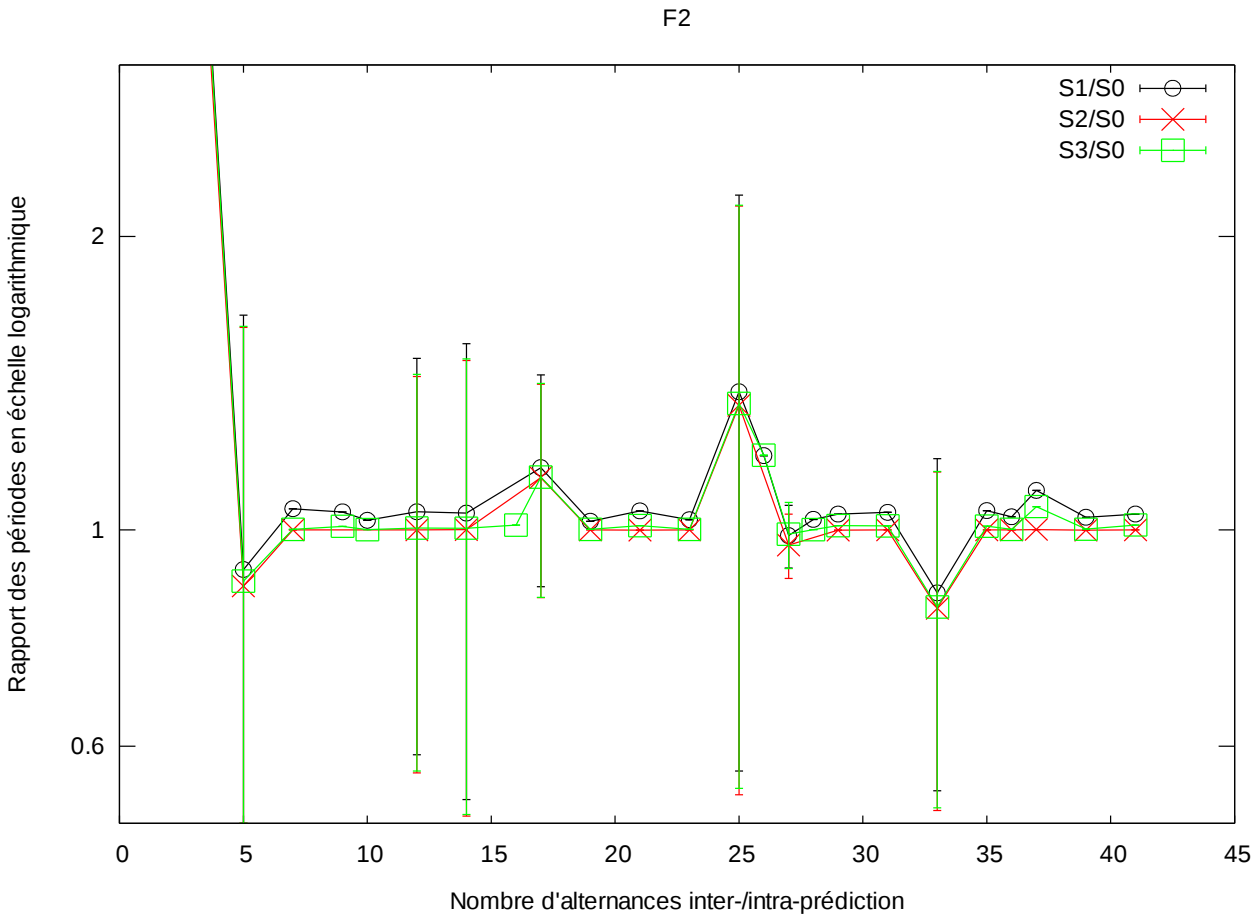


FIGURE 6.8 – *Rapports des périodes entre deux images décodées successives en fonction du nombre d'alternances entre inter- et intra-prédiction. Taille des files inter-acteurs : 2.*

Dans le cas de F3, l'écart entre S1 et S2 reste du même ordre qu'avec F2 ; en revanche, l'écart se creuse avec S3. C'est en effet là que les paramètres indicatifs utilisés uniquement par S3 montrent le plus leur impact. Dans certains cas S3 est nettement meilleur, par exemple pour 7 et 18 alternances, dans d'autres il est clairement moins bon, comme pour 37 alternances. Les valeurs médianes pour les trois stratégies sont sensiblement les mêmes que pour F2. D'autre part, et contrairement à ce qui était attendu, les résultats obtenus ne permettent pas de montrer un avantage compétitif de S3 lorsque le nombre d'alternances augmente, ce qui constitue pourtant un scénario favorable à l'exploitation des paramètres liés au type de prédiction. L'explication tient à la très forte variabilité de l'application qui n'a pu être capturée que de façon parcellaire *via* lesdits paramètres. En effet, ils ne tiennent compte que de l'effet du type de prédiction au niveau de l'acteur `ipred`, mais ne disent rien des facteurs de variabilité des autres acteurs, ni de ceux d'`ipred` indépendants du type de prédiction. Une analyse plus approfondie de l'application serait vraisemblablement nécessaire.

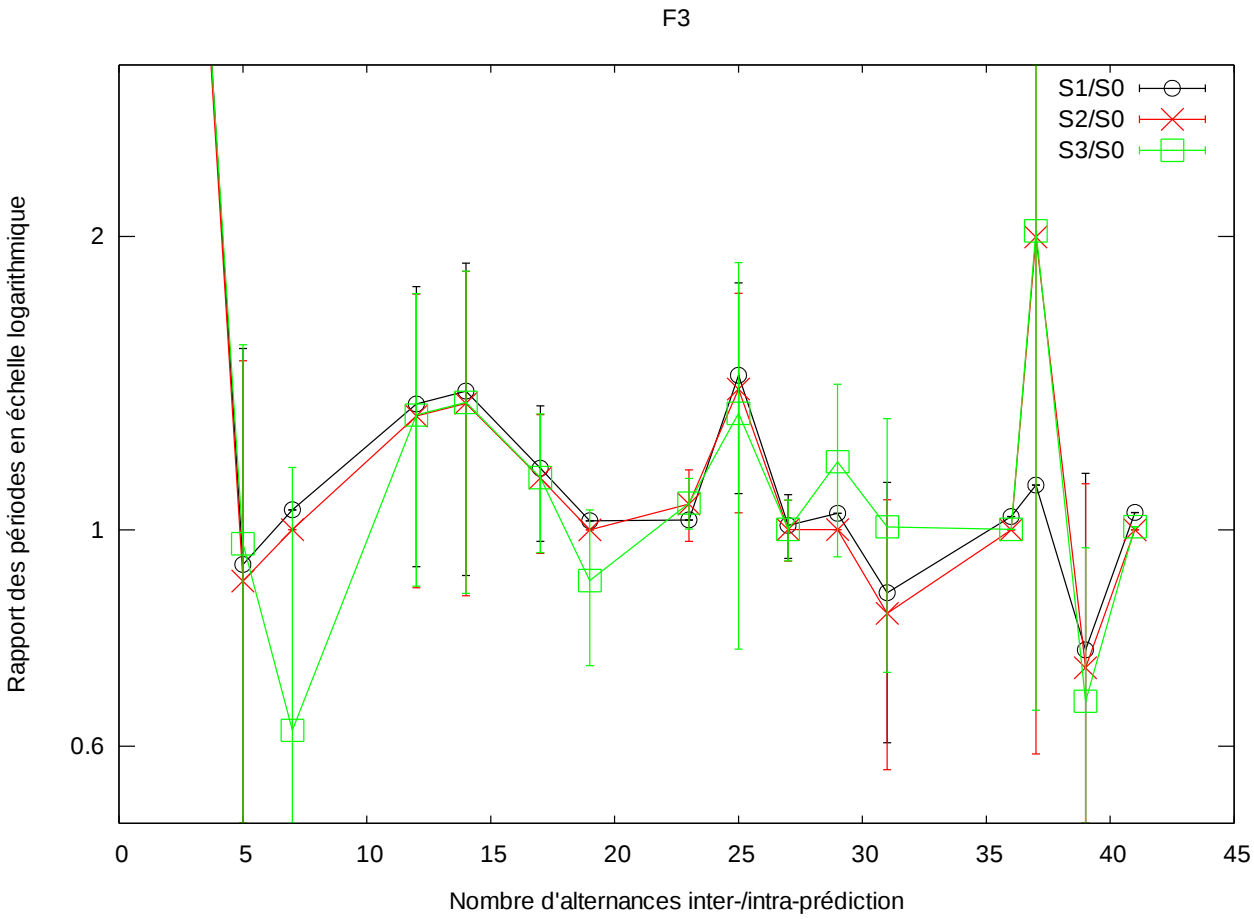


FIGURE 6.9 – *Rapports des périodes entre deux images décodées successives en fonction du nombre d'alternances entre inter- et intra-prédiction. Taille des files inter-acteurs : 3.*

Néanmoins, si l'on considère S3 comme une stratégie auto-ajustable variant sur une échelle dont S1 et S2 seraient les deux extrêmes, on remarque qu'elle a tendance à s'approcher de la plus adaptée. Ainsi, s'il n'est pas évident, à priori, de déterminer quelle politique est la meilleure, l'introduction des paramètres indicatifs apporte des éléments de réponse. Dans le cas de F2, on voit par exemple clairement que S3 tend vers S2, qui est alors la meilleure stratégie dynamique. Cela justifie donc le recours aux paramètres indicatifs.

6.3 Conclusion

Ce chapitre a présenté une justification théorique de l'utilité des paramètres indicatifs et un jeu d'expérimentations et de mesures visant à évaluer l'efficacité du modèle d'exécution, de la gestion dynamique et des paramètres indicatifs. Il en ressort que le coût brut, à la fois spatial et temporel, est très modeste. La comparaison de plusieurs stratégies d'ordonnancement avec différentes tailles de files inter-acteurs a permis d'illustrer de façon limitée les gains potentiels apportés par le dynamisme, mais aussi de montrer les bénéfices des paramètres indicatifs quant au choix d'une politique adaptée. Néanmoins, des travaux supplémentaires sont nécessaires pour en montrer les gains nets en pratique[35].

35 L'accès aux outils et à la plateforme de simulation n'étant définitivement plus possible depuis la fin du contrat avec STMicroelectronics, toutes les expériences souhaitées n'ont pu être menées.

Une étude plus approfondie du décodeur H.264 ainsi que d'autres applications telles que des algorithmes utilisés dans les télécommunications pourrait être menée. Cela permettrait de mettre en lumière tous les facteurs de variabilité prépondérants et de les capturer à l'aide de paramètres indicatifs. En reprenant le protocole expérimental présenté précédemment et éventuellement en introduisant de nouvelles politiques d'ordonnancement, il serait alors possible d'évaluer de manière plus fidèle l'efficacité des paramètres indicatifs. De plus, des expériences sur un plus grand nombre d'acteurs logiciels s'avèrent nécessaires pour tirer pleinement parti du modèle d'exécution.

Conclusion

La question de l'exécution efficace d'applications à flux de données est délicate, y compris sur des architectures conçues à cet effet telles que STHORM. À cet égard, l'hybridité est à la fois un atout et un frein : un atout car le support matériel d'un mécanisme de transfert de données par flux se prête bien à ce type d'application ; un frein car la programmabilité de ces architectures est rendue difficile par le manque de modèles et d'outils dédiés. Parmi les modèles existants, la vaste classe de ceux à flot de données permet une description efficace des applications visées et une projection facilitée sur le matériel. Néanmoins, malgré les nombreuses variantes déjà proposées (SDF, PSDF, etc.), aucune ne répond convenablement à la fois aux besoins des plateformes hybrides récentes et à ceux des applications à flux de données complexes comme les codecs vidéos (cf. chapitre 2). Cette complexité se traduit par un besoin croissant en compacité et en expressivité du modèle, mais surtout en un haut degré de dynamisme qui doit nécessairement être pris en compte à l'exécution sous peine de performances médiocres – et donc de conditions de visionnage dégradées pour l'utilisateur, dans le cas du décodage vidéo. Ce fort dynamisme est causé par des reconfigurations fréquentes liées au caractère imprédictible des données traitées, et qui induisent de fortes variations en termes de durée d'exécution. Ces variations ont un impact non négligeable sur les décisions d'ordonnancement et doivent donc être en première ligne de cette prise en compte. Les contributions présentées dans cette thèse s'inscrivent dans cet axe.

La première d'entre elles répond à la nécessité d'un modèle d'exécution flexible à même de capturer la variabilité intrinsèque des applications à flux de données dynamiques, d'une part, et de s'adapter à des architectures hybrides dont le modèle a été décrit au chapitre 1, d'autre part. Le modèle proposé au chapitre 2 est fondé sur des théories qui ont fait leurs preuves pour les algorithmes de traitement du signal dans les systèmes embarqués – celles des réseaux de processus et des flots de données – et en fait une composition permettant de tirer parti des avantages de chacune. Ce modèle paramétrique offre une expressivité et une compacité élevées qui rendent possible la représentation des applications précitées. Mais la nouveauté majeure réside dans l'introduction des paramètres indicatifs porteurs d'informations à destination de l'ordonnanceur concernant les durées d'exécution et les quantités de données transférées lorsque celles-ci sont variables et imprédictibles statiquement. Ces indications permettent un ajustement dynamique de l'ordonnancement pour réagir à ces variations. Par ailleurs, une classification étendue des paramètres prenant en compte les différents types de reconfigurations possibles a été présentée.

La seconde contribution concerne spécifiquement la prise en compte des contraintes mémorielles qui pèsent lourdement sur l'exécution du logiciel embarqué. En particulier, il a été montré qu'elles jouent un rôle prépondérant dans l'efficacité de la stratégie d'ordonnancement adoptée. C'est pourquoi le chapitre 3 a exposé une extension des algorithmes d'ordonnancement de liste clas-

siques visant à la prise en compte de ces contraintes. Cette contribution est multiple. Dans un premier temps, elle propose une modélisation de l'application sous forme de graphe de tâches dont certaines sont spécifiquement dédiées à une description précise de la quantité de mémoire requise à chaque instant de l'exécution. Cette description permet de déduire la borne inférieure de la capacité mémorielle de la plateforme cible permettant de garantir une exécution complète, contrairement aux algorithmes de base qui mènent invariablement à des impasses en cas de pression élevée. De plus, les expérimentations menées ont permis de montrer qu'en plus de produire des ordonnancements licites toutes les fois où cela est possible, on observe des accélérations qui, dans certains cas, dépassent 20 %.

Enfin, la troisième contribution vient soutenir le modèle d'exécution de la première en apportant un support logiciel complet détaillé au chapitre 5 et s'articulant autour de trois axes. Le premier concerne la gestion dynamique des acteurs et offre la réactivité à même de répondre au mieux au dynamisme de l'application grâce à sa structure distribuée et à son support intrinsèque de l'hybridité de la plateforme. Le second, étroitement lié au premier, introduit un ordonnanceur, lui aussi distribué pour une meilleure réactivité, dont l'activité est décomposée en trois phases de manière à permettre aisément le branchement de la politique d'ordonnancement souhaitée par l'utilisateur. Le dernier vise à la mise en œuvre des paramètres indicatifs en introduisant l'interface et les mécanismes nécessaires à leur interprétation par l'ordonnanceur.

Les expérimentations menées sur des applications réelles avec les outils de STMicroelectronics pour la plateforme STHORM (cf. chapitre 4) en vue d'évaluer l'efficacité d'une telle approche ont abouti aux résultats présentés au chapitre 6. La première conclusion est que le coût à la fois spatial et temporel est faible en regard des contraintes qui pèsent sur les systèmes embarqués visés. La seconde est que, dans les exemples limités qui ont pu être étudiés pendant la durée de la thèse, il n'a pas été possible de montrer que le dynamisme apportait un gain net en termes de performances. En revanche, l'utilité des paramètres indicatifs a pu être illustrée en comparant des politiques d'ordonnancement les prenant en compte ou non : ils permettent de tendre automatiquement vers la stratégie la plus adaptée. Des travaux complémentaires seraient néanmoins nécessaires pour mesurer précisément l'apport en efficacité.

Perspectives

La tendance étant à la complexification des applications, en particulier pour les algorithmes de décodage vidéo (cf. chapitre 1), la nécessité croissante de recourir à des modèles tels que celui présenté précédemment est patente. Il serait donc profitable que les travaux menés au cours de cette thèse soient approfondis et étendus de différentes façons. En premier lieu, la priorité serait de reprendre l'application H.264, mais en se fondant non plus sur l'implémentation propriétaire et difficile à prendre en main de STMicroelectronics, mais sur une version ouverte facilitant la collaboration de divers partenaires académiques comme celle développée par les contributeurs du projet Orcc[36]. Dans un second temps, il serait nécessaire de mener des expériences sur un plus grand nombre d'applications : le codec HEVC serait le meilleur candidat dans le domaine du décodage vidéo puisqu'il est amené à en devenir le principal représentant dans un avenir proche, supplantant H.264. Il serait aussi utile de s'intéresser aux algorithmes de télécommunications, en

36 https://github.com/orcc

particulier pour la téléphonie mobile, qui se prêtent bien à une modélisation à flot de données [77], [78]. Quelle que soit l'application, les outils de STMicroelectronics n'étant disponibles qu'en interne, il serait nécessaire d'adapter le logiciel système – qui comprend notamment le support du modèle d'exécution avec les paramètres indicatifs – d'abord à l'infrastructure d'Orcc et ensuite à la cible choisie, qui ne pourra donc plus être STHORM.

Une part importante de l'effort de développement doit dans tous les cas être allouée à la compréhension et à la modélisation de l'application, l'idéal étant que les architectes travaillent de concert avec les développeurs en ayant une bonne connaissance. Il serait alors possible de modéliser très finement son comportement, en particulier les facteurs de dynamisme (variabilité et reconfiguration), et de les capturer à l'aide de paramètres indicatifs. Par la suite, le protocole expérimental pourrait rester analogue à celui adopté au chapitre 6, en testant davantage de stratégies d'ordonnancement puis en recueillant des métriques supplémentaires (taux d'occupation des processeurs, latence des communications avec prise en compte de la contention, etc.). Ces deux derniers points impliquent néanmoins respectivement que le nombre de filtres logiciels soit suffisant – un ratio d'un filtre logiciel pour un filtre matériel paraissant raisonnable – et que l'environnement de test le permette – ce qui n'était par exemple pas le cas du simulateur utilisé pour STHORM. En outre, pour obtenir des résultats significatifs, il serait nécessaire d'avoir accès aux temps d'exécution des filtres matériels, soit par le biais d'un modèle plus poussé que celui présenté au chapitre 5, soit en réalisant les mesures sur du matériel réel.

Du point de vue de l'architecture, l'hybridité est amenée à se développer comme une forme d'hétérogénéité avancée permettant de mettre l'efficacité du matériel au service non seulement des applications à flux de données, mais aussi de toutes celles requérant une puissance de calcul à la fois élevée et déterministe. Cette évolution ne se fera pas sans que soient relevés un ensemble de défis liés notamment à la cohabitation entre l'espace Von Neumann et l'espace flot de données, au nombre desquels la question de la communication des données entre les deux mondes soulevée au chapitre 5. Par ailleurs, une difficulté importante à surmonter est celle de la réalisation des blocs matériels par l'utilisateur, opération chronophage lorsqu'elle est faite directement en RTL. Il est par conséquent impératif de développer des outils autorisant le passage efficace d'une description logicielle dans un langage tel que le C à une implémentation matérielle. C'est pourquoi les méthodes de synthèse de haut niveau joueront vraisemblablement un rôle important à l'avenir.

En conclusion, quelles que soient les mutations à venir dans le paysage de l'électronique multimédia, il sera de la plus haute importance d'accentuer l'intégration entre logiciel et matériel. Cela passe avant tout par une plus grande collaboration entre architectes, concepteurs, développeurs et utilisateurs, à toutes les étapes du cycle d'élaboration. Mais cela passe aussi par le recours à des méthodes et outils adaptés aux spécificités de l'embarqué, qui ne soient pas de simples transpositions de ce qui se fait en informatique généraliste. L'électronique embarquée reste un domaine de recherche à part entière qui ne saurait se résumer à une combinaison d'électronique numérique et d'informatique, comme l'a montré l'expérience de cette thèse.

Bibliographie

[1] Didier Forray, “Du Walkman à l’iPod, 30 ans de balade,” *01net*, 07-Jan-2009. [Online]. Available: http://www.01net.com/editorial/503883/du-walkman-a-lipod-30-ans-de-balade/. [Accessed: 26-Nov-2014].

[2] Michel Desmurget, “Les conséquences que le temps fou passé devant des écrans a sur le cerveau,” 03-Aug-2014.

[3] S. Saponara, K. Denolf, G. Lafruit, C. Blanch, and J. Bormans, “Performance and Complexity Co-evaluation of the Advanced Video Coding Standard for Cost-Effective Multimedia Communications,” *EURASIP J. Adv. Signal Process.*, vol. 2004, no. 2, p. 214371, Mar. 2004.

[4] J. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, “Comparison of the Coding Efficiency of Video Coding Standards #x2014;Including High Efficiency Video Coding (HEVC),” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1669–1684, décembre 2012.

[5] J. Gorin, M. Raulet, and F. Prêteux, “MPEG Reconfigurable Video Coding: From specification to a reconfigurable implementation,” *Signal Process. Image Commun.*, vol. 28, no. 10, pp. 1224–1238, Nov. 2013.

[6] J. Eker and J. Janneck, “CAL language report,” *Univ. Calif. Berkeley Tech Rep UCBERL M*, vol. 3, 2003.

[7] A. Salkintzis and N. Passas, *Emerging Wireless Multimedia: Services and Technologies*. John Wiley & Sons, 2005.

[8] I. E. G. Richardson, *H.264 and MPEG-4 video compression: video coding for next generation multimedia*. Chichester; Hoboken, NJ: Wiley, 2003.

[9] A. K. Khan and H. Jamal, “The Intra Prediction in H.264,” in *Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics*, T. Sobh, K. Elleithy, A. Mahmood, and M. A. Karim, Eds. Springer Netherlands, 2008, pp. 11–15.

[10] I. Richardson, *White Paper: H. 264/AVC Inter Prediction*. Vcodex, 2002.

[11] S.-Z. Wang, T.-A. Lin, T.-M. Liu, and C.-Y. Lee, “A new motion compensation design for H.264/AVC decoder,” in *IEEE International Symposium on Circuits and Systems, 2005. ISCAS 2005*, 2005, pp. 4558–4561 Vol. 5.

[12] M. Samek, “CONTRIBUTING EDITORS-The Embedded Angle-The Embedded Mindset-Innovation in skinning the Heap Cat.,” *CC Users J.*, pp. 39–45, 2003.

[13] J. L. Hennessy, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5 edition. Waltham, MA: Morgan Kaufmann, 2011.

[14] W. Sheng, S. Schürmans, M. Odendahl, M. Bertsch, V. Volevach, R. Leupers, and G. Ascheid, “A compiler infrastructure for embedded heterogeneous MPSoCs,” *Parallel Comput.*, vol. 40, no. 2, pp. 51–68, février 2014.

[15] John R. Levine, *Linkers and Loaders*. San Francisco: Morgan Kaufmann, 1999.

[16] U. Drepper, “How to write shared libraries,” *Retrieved Jul*, vol. 16, p. 2009, 2006.

[17] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, “Hybrid Dataflow/von-Neumann Architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp.

1489–1509, Jun. 2014.

[18] M. Dehyadegari, A. Marongiu, M. R. Kakoee, L. Benini, S. Mohammadi, and N. Yazdani, “A tightly-coupled multi-core cluster with shared-memory HW accelerators,” in *2012 International Conference on Embedded Computer Systems (SAMOS)*, 2012, pp. 96–103.

[19] P. Chen, L. Zhang, Y.-H. Han, and Y.-J. Chen, “A General-Purpose Many-Accelerator Architecture Based on Dataflow Graph Clustering of Applications,” *J. Comput. Sci. Technol.*, vol. 29, no. 2, pp. 239–246, Mar. 2014.

[20] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, “Tartan: Evaluating Spatial Computation for Whole Program Execution,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2006, pp. 163–174.

[21] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, “DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing,” *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep. 2012.

[22] J. B. Dennis, “First version of a data flow procedure language,” in *Programming Symposium*, P. B. Robinet, Ed. Springer Berlin Heidelberg, 1974, pp. 362–376.

[23] E. Lee, “Consistency in dataflow graphs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 2, pp. 223–235, avril 1991.

[24] T. M. Parks, “Bounded scheduling of process networks,” University of California, 1995.

[25] S. A. Neuendorffer, “Actor-oriented metaprogramming,” University of California, Berkeley, 2005.

[26] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for DSP systems,” *IEEE Trans. Signal Process.*, vol. 49, no. 10, pp. 2408–2421, Oct. 2001.

[27] P. Fradet, A. Girault, and P. Poplavkoy, “SPDF: A schedulable parametric data-flow MoC,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, 2012, pp. 769–774.

[28] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, “BPDF: A statically analyzable dataflow model with integer and boolean parameters,” in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2013, pp. 1–10.

[29] J. T. Buck and E. A. Lee, “Scheduling dynamic dataflow graphs with bounded memory using the token flow model,” in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, 1993, vol. 1, pp. 429–432.

[30] J. T. Buck, “Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams,” in *1994 Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers, 1994*, 1994, vol. 1, pp. 508–513 vol.1.

[31] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” in *Information Processing ’74: Proceedings of the IFIP Congress*, J. Rosenfeld, Ed. North-Holland, 1974, pp. 471–475.

[32] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[33] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, “Cycle-static dataflow,” *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, février 1996.

[34] S. Stuijk, “Predictable mapping of streaming applications on multiprocessors,” 2007.

[35] *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[36] M. Geilen and T. Basten, “Requirements on the execution of Kahn process networks,” in *Programming languages and systems*, Springer, 2003, pp. 319–334.

[37] G. E. Allen, P. E. Zucknick, and B. L. Evans, “A Distributed Deadlock Detection and Resolution Algorithm for Process Networks,” in *IEEE International Conference on Acoustics,*

Speech and Signal Processing, 2007. ICASSP 2007, 2007, vol. 2, pp. II–33–II–36.

[38] B. Jiang, E. Deprettere, and B. Kienhuis, "Hierarchical run time deadlock detection in process networks," in *IEEE Workshop on Signal Processing Systems, 2008. SiPS 2008*, 2008, pp. 239–244.

[39] A. G. Olson and B. L. Evans, "Deadlock detection for distributed process networks," in *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on*, 2005, vol. 5, pp. v–73.

[40] E. L. Lawler, J. K. Lenstra, A. H. Rinnooy Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," *Handb. Oper. Res. Manag. Sci.*, vol. 4, pp. 445–522, 1993.

[41] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP," in *Global Telecommunications Conference, 1989, and Exhibition. Communications Technology for the 1990s and Beyond. GLOBECOM'89., IEEE*, 1989, pp. 1279–1283.

[42] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[43] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[44] S. Ha, "Compile-time scheduling of dataflow program graphs with dynamic constructs," University of California, Berkeley, 1992.

[45] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng, "Comparative evaluation of the robustness of dag scheduling heuristics," in *Grid Computing*, 2008, pp. 73–84.

[46] Z. Shi and J. J. Dongarra, "Scheduling workflow applications on processors with different capabilities," *Future Gener. Comput. Syst.*, vol. 22, no. 6, pp. 665–675, May 2006.

[47] H. Arabnejad and J. G. Barbosa, "List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.

[48] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *Parallel Distrib. Syst. IEEE Trans. On*, vol. 4, no. 2, pp. 175–187, 1993.

[49] R. Sethi, "Complete Register Allocation Problems," *SIAM J. Comput.*, vol. 4, no. 3, pp. 226–248, Sep. 1975.

[50] P. Briggs, "Register allocation via graph coloring," Rice University, 1992.

[51] G. Chaitin, "Register Allocation and Spilling via Graph Coloring," *SIGPLAN Not*, vol. 39, no. 4, pp. 66–74, avril 2004.

[52] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000, pp. 109–114.

[53] J. W. H. Liu, "On the Storage Requirement in the Out-of-core Multifrontal Method for Sparse Factorization," *ACM Trans Math Softw*, vol. 12, no. 3, pp. 249–264, Sep. 1986.

[54] L. Marchal, O. Sinnen, and F. Vivien, "Scheduling Tree-Shaped Task Graphs to Minimize Memory and Makespan," in *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, 2013, pp. 839–850.

[55] J. Herrmann, L. Marchal, and Y. Robert, "Memory-aware list scheduling for hybrid platforms," Feb. 2014.

[56] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Mar. 1991.

[57] J. Gilbert, T. Lengauer, and R. Tarjan, "The Pebbling Problem is Complete in Polynomial

Space," *SIAM J. Comput.*, vol. 9, no. 3, pp. 513–524, août 1980.

[58] I. Auge, F. Petrot, F. Donnet, and P. Gomez, "Platform-based design from parallel C specifications," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 12, pp. 1811–1826, décembre 2005.

[59] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia Development Made Easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, New York, NY, USA, 2013, pp. 863–866.

[60] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software Code Generation for the RVC-CAL Language," *J. Signal Process. Syst.*, vol. 63, no. 2, pp. 203–213, Jun. 2009.

[61] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs," *IEEE Trans. Ind. Inform.*, vol. 9, no. 1, pp. 527–545, février 2013.

[62] L. Ferro, "Vérification de propriétés logico-temporelles de spécifications SystemC TLM," Université de Grenoble, 2011.

[63] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 606–609.

[64] M. H. Wiggers, M. J. Bekooij, and G. J. Smit, "Monotonicity and run-time scheduling," in *Proceedings of the seventh ACM international conference on Embedded software*, 2009, pp. 177–186.

[65] H. Lee, W. Che, and K. Chatha, "Dynamic scheduling of stream programs on embedded multi-core processors," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2012, pp. 93–102.

[66] E. Garcia, D. Orozco, R. Pavel, and G. R. Gao, "A Discussion in Favor of Dynamic Scheduling for Regular Applications in Many-core Architectures," 2012, pp. 1591–1600.

[67] Z. Zhou, K. Desnos, M. Pelcat, J.-F. Nezan, W. Plishker, and S. S. Bhattacharyya, "Scheduling of parallelized synchronous dataflow actors," in *2013 International Symposium on System on Chip (SoC)*, 2013, pp. 1–10.

[68] G. Liu, Y. He, L. Guo, and F. Qi, "Static Scheduling of Synchronous Data Flow onto Multiprocessors for Embedded DSP Systems," 2011, pp. 338–341.

[69] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications," in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 189–198.

[70] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a Data-flow Programming Model Onto Heterogeneous Platforms," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, New York, NY, USA, 2012, pp. 61–70.

[71] H.-H. Wu, C.-C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A Model-Based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs," 2011, pp. 70–81.

[72] J. Boutellier, C. Lucarz, V. M. Gomez, M. Mattavelli, and O. Silvén, "Multiprocessor Scheduling of Dataflow Programs within the Reconfigurable Video Coding Framework," in *Algorithm-Architecture Matching for Signal and Image Processing*, vol. 73, G. Gogniat, D. Milojevic, A. Morawiec, and A. Erdogan, Eds. Dordrecht: Springer Netherlands, 2011, pp. 237–251.

[73] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet, "Efficient multicore scheduling of dataflow process networks," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, 2011, pp. 198–203.

[74] H. Yviquel, E. Casseau, M. Raulet, P. Jaaskelainen, and J. Takala, "Towards run-time actor

mapping of dynamic dataflow programs onto multi-core platforms," in *2013 8th International Symposium on Image and Signal Processing and Analysis (ISPA)*, 2013, pp. 732–737.

[75] H. Yviquel, A. Sanchez, P. Jaaskelainen, J. Takala, M. Raulet, and E. Casseau, "Efficient software synthesis of dynamic dataflow programs," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014, pp. 4988–4992.

[76] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst. TOPLAS*, vol. 5, no. 2, pp. 190–222, 1983.

[77] J. Heulot, J. Boutellier, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Applying the adaptive Hybrid Flow-Shop scheduling method to schedule a 3GPP LTE physical layer algorithm onto many-core digital signal processors," in *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2013, pp. 123–129.

[78] H. Kee, S. S. Bhattacharyya, I. Wong, and Y. Rao, "FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques," presented at the ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings, 2010, pp. 1510–1513.

[79] J. Epstein, *The Yale Book of Quotations*, Illustrated edition edition. New Haven: Yale University Press, 2006.

[80] M. Wipliez, "Infrastructure de compilation pour des programmes flux de données," INSA de Rennes, 2010.

[81] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell," *IEEE Signal Process. Mag.*, vol. 27, no. 3, pp. 159–167, May 2010.

[82] I. Richardson, *White Paper: An Overview of H. 264 Advanced Video Coding*. Vcodex, 2007.